UiO : Department of Informatics
University of Oslo

Adaptation trigger mechanism

Context-driven adaptation trigger evaluated with simulated multimedia server and player

Goran Karabeg
Master's Thesis
Spring 2014

Adaptation trigger mechanism

Goran Karabeg

3rd June 2014

Abstract

The common way of consuming multimedia content is by using pre-installed client applications which request a multimedia stream from the server. If the host environment of the client application changes, e.g. bandwidth drops or CPU is overburden, the application triggers content adaptation and server responds with the adapted video stream.

We assist client applications by automating the adaptation trigger process thus relieving applications of that responsibility. We design, implement and evaluate the adaptation trigger mechanism (ATM) to examine if such a mechanism can aid legacy client applications.

We address three questions in this thesis:

1. Can the ATM help legacy client applications by reacting to context change and requesting an adapted video stream for the players by keeping the reaction time below a tolerable delay?

2. Can the same ATM serve to many different client applications at the same time and achieve the same efficiency?

3. Can we build ATM as an easily extendible dynamic library while still achieve same efficiency in regards to the run-time execution?

While the answers for the last two question are confirmed positive, the answer for the first question still remains open, because the design of clients applications is limiting our ATM to aid in the content adaptation.

Contents

List of Figures

List of Tables

Acknowledgements

The work in this thesis would not have been possible without the kind support and help from my supervisors, PhD student Francisco Velázquez and Professor Thomas Plagemann. I would like to express my gratitude for the time they invested in the supervision, guidance and contributions to my work.

I would also like to thank to PhD Florian Wamser from the Informatics Department at University of Wuerzburg, Germany for providing guidelines for YouTube monitoring tool YoMo.

Last but not least, I would like to thank my family, especially to my wife Mirela who supported me completely in my writing.

Chapter 1

Introduction

In this chapter we look at the motivation for the master thesis. We discuss the main goals and the problem statements that the thesis tries to answer. Lastly we describe methods and define scope of the thesis.

1.1 Motivation

People today own many devices that have access to the Internet and can consume multimedia content. The devices can be a TV set, a Personal Computer (PC), a smartphone, a tablet and many other. The traditional way of consuming multimedia content is by using pre-installed applications that have access to multimedia content delivery networks (CDN) where they request the multimedia content and stream it to the device. A few well-known multimedia applications are Youtube, Netflix, Vimeo and Dailymotion. They all have client applications specially designed for different devices with different operative systems (OS). They also have different versions of the application in different browsers with specific codecs support. Applications make use of a user profile to allow users to filter the multimedia content and save the video and video position for later viewing. A user can continue to watch the video at another time and on another device after the user has logged on. If a user wants to switch the device while streaming video, he has to manually stop it on one device and start it on the other.

We envision a way of consuming multimedia content on different devices by allowing an application to migrate from one device to another instantly. The application migration takes effect while the client application is streaming without service interruption or annoying delay for the user of the application.

The work in this master thesis is inspired by a project called TRAMP Real-time Application Mobility Platform. TRAMP [23] is a research project at the Distributed Multimedia Systems group at the University of Oslo. The project's goal is to develop a seamless fine-grained migration system for real-time applications which utilize available multimedia devices in the best possible way. The proposed solution of TRAMP project is a middleware which offers an API to facilitate the development of migration

of fine-grained multimedia applications.

TRAMP is an effort towards ubiquitous computing. TRAMP's main claim is that seamless migration in ubiquitous computing increases Quality of Experience (QoE) of users of multimedia applications.

As a step toward ubiquitous use of multimedia applications, we design, implement and evaluate an adaptation trigger mechanism (ATM) which is a component used in the multimedia content adaptation process. When the context of the client application changes, the multimedia content needs to be adapted in order to be accessible. Our adaptation trigger facilitates the process by listening to context changes, reading the context parameters and communicating the parameters to a multimedia server and to a client application. The ATM handles communication between multimedia server and client. The efficient automated communication is the main motivation for this thesis.

1.2 Goal

In this thesis, we develop a general approach for triggering adaptation process when the context changes. The meaning of "context" is described in Section 2.1. Most of the client applications in use today are responsible for handling the host environment context changes such as available networks bandwidth. They adapt the multimedia content by requesting a stream with a different bitrate from a multimedia server. The ATM aims to automate reaction to context changes thus relieving client applications from that responsibility.

The major goal of this thesis is the design, implementation and evaluation of the ATM with the focus on efficiency, extendibility and maintainability. The ATM is designed as a dynamic library which is loaded by a background process. The ATM handles context parameters and aid client applications to get the optimal multimedia stream. The client applications can be many and of various type. They can also consume stream from various types of multimedia servers.

In order to be efficient the ATM has to react fast enough so the user's QoE does not decrease. It has to be easily extendible and maintainable, so that logic for supporting other types of client applications and multimedia servers can be easily added.

1.3 Problem statement

Users tolerate about 100 ms of delay in highly interactive tasks like video conferencing [30]. During this timespan the users might experience video delivery problems while multimedia content is adapting. This timespan includes the ATM time to react and the time for the multimedia streaming server to respond.

In this thesis we want to answer the main questions related to tolerable efficiency and extendibility of the ATM as a part of the middleware.

1. Traditional video applications are either not reacting on the context changes or are reacting only on a limited set of parameters, e.g on bandwidth changes or CPU usage. Can the ATM help legacy players by reacting to context change and requesting an adapted video stream for the players by keeping the reaction time below a tolerable delay?

2. Can the same ATM serve to many different applications at the same time and achieve the same efficiency?

3. Can we build ATM as an easily extendible dynamic library while still achieve same efficiency in regards to the run-time execution?

Answers for these questions are the main subject for the conclusion part of the thesis.

1.4 Methods

The methods used in the thesis are based on two core assumptions:

- multimedia content providers such as YouTube offer dynamic content adaptation which are triggered by a set of URL parameters from the ATM call, and

- most widely used multimedia components have an accessible API for controlling media source link which ATM can use.

The methods used in the thesis are design, implementation and evaluation. The thesis is of practical nature where we try to find the best way to implement an ATM module. For every class or software module used with ATM we investigate available systems and software, compare the alternatives and use the ones that best fit the ATM requirements. The empirical results are gathered in test runs into data sets and evaluated by examining average value and variance.

1.5 Scope and time limitations

We want the ATM to include the logic to communicate with different multimedia servers so that the video players get the stream through the ATM. There are however situations where this is not feasible and a video player has a direct communication link with the multimedia server. Considering these situations the scope is divided into two parts. The first part involves ATM development working with existing commercial video players and multimedia servers. The second part involves working with a simulated player and server.

The scope of the first part is to design and implement the ATM that dynamically reacts to context changes and communicate the changes to an adaptation mechanism. Within that scope is development of code wrappers for conventional video player components. The second part has a focus on the design and implementation of a simulated server and player and

extending of ATM logic.

In Section 3.1, we present two use cases where adaptation is triggered by the ATM. In both cases, the ATM need to be aware of context changes by listening to available device sensors and device drivers changes. Because of time limitations, the actual context sensing is left for future work, and we simulate the context parameters by reading from a customized file. The portability issues related to different operative systems for personal computers and mobile phones are also out of the scope of this thesis.

1.6 Structure of thesis

The thesis is divided in 7 chapters.

Chapter 1. Introduction This chapter states motivation, goal, problem statement, methods and scope of the thesis.

Chapter 2. Background In this chapter we define context and context-aware adaptation for this theses. We also investigate latest trends in streaming, multimedia content adaptation and research on the dynamic content adaptation.

Chapter 3. Planning the project This chapter presents requirements and the design of ATM, video player wrappers and inter-process communication.

Chapter 4. Developing the project In this chapter, we describe the implementation process, discuss implementation decisions and findings. At the end of the chapter we turn our attention to limited options available for ATM to handle communication between the multimedia server and the client applications.

Chapter 5. Simulation design and development This chapter describes design and implementation of the simulated server and player together with the changes in the initial design of ATM.

Chapter 6. Evaluation This chapter defines the ATM test bed, present test results and evaluates the results. It also includes discussion of comparing the results to the YouTube buffer management.

Chapter 7. Conclusion Finally the last chapter presents conclusion, and discuss contribution and future work.

Chapter 2

Background

This chapter gives an overview of the meaning of the context and context-aware content adaptation. We discuss the latest trends in the conventional streaming and look into research projects regarding the multimedia content adaptation.

2.1 Context-aware multimedia content adaptation

Most of the available multimedia content on the Internet today is made for personal computers with access to the broadband high speed Internet. This content is not suitable for devices with limited resources as small size displays, limited processing power and variable network bandwidth. It has become important to provide personalization of multimedia information according to users personal preference. As example a user can express his interest based on a TV show genre to filter out multimedia content before presenting it to the user. In addition to user preferences, user context like location (home, work, restaurant, etc.), current activity (driving, jogging, in the meeting, travelling, etc.), time of the day (morning, evening, etc.) and other possible situation, have been increasingly used in multimedia content delivery.

As a result much research has been dedicated to assist context-aware adaptation of original multimedia content to suitable content. To really grasp the context-aware multimedia content adaptation it is important to understand the meaning of the context. Day explains in his paper [17] that *"Context is any information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves."* Jain and Sinha explain in their paper [31] that adapting multimedia content without context is meaningless.

The primary goal of multimedia content adaptation is to increase multimedia accessibility. Lei and Goerganas explain *"Context-based media adaptation technology is mainly concerned with selecting different qualities of single media types or selecting different media types, and delivering information to different context, such as location, device capability, network bandwidth, user preference, etc. With media adaptation technology, multimedia information can be filtered,*

transformed, converted or reformed to make it universally accessible by different devices, and to provide personalized multimedia content to different users" [34].

There exist many distinctive ways of multimedia content adaptation intended to be used with different goals and situations. Multiple adaptations can be combined to arrive at the desired format for a given context. Lei and Goerganas [34] categorize multimedia content adaptation based on different perspectives.

- *"**According to target context** :*
  - ***Adaptation to technical infrastructure**: Multimedia content has to be adapted according to device capabilities and network.*
  - ***Adaptation to user context**: Multimedia content adaptation is done according to user preferences of information details.*
- ***According to when adaptation is created***
  - ***Static adaptation**: Multimedia is preprocessed into different alternatives - quality, bitrate, formats*
  - ***Dynamic adaptation**: Multimedia is process at time requested (at runtime).*
- ***According to media type***
  - ***Single media element adaptation**: For example transforming image to different formats as GIF or JPEG.*
  - ***Cross-media adaptation (modality)**: Transforming a video into series of images, for devices not capable of rendering video.*
- ***According to abstract adaptation level***
  - ***Semantic adaptation**: Selective process of extracting information that user is interested in.*
  - ***Physical adaptation**: Physical adaptation of media defined as combination of conversion, scaling and distillation process guided by media format and physical QoS."*

The multimedia content adaptation can take place generally in three different locations. Adzic explain [2] that the locations are:

- **Server-side adaptation**. With this approach a multimedia content provider has control of adaptation. Using this approach has several benefits where one is a good network utilization by using only needed bandwidth for given device and another is that content providers can protect their copyrights. The drawback is that the servers are overloaded with both content delivery and content adaptation for many types of devices.

- **Client-side adaptation**. This approach has the benefit that a user is in charge of all adaptation, can define preferences and types for adaptation. The drawback is that some devices cannot perform adaptation, so if the server is not adapting the multimedia content then the users device cannot consume it. Other drawbacks are poor bandwidth utilization and need for extra software (plugins) on clients, which makes adaptation not transparent to the users.

- **Proxy adaptation**. This approach is most often favourable because of the several benefits it provides. It is implemented independently of multimedia content providers and has the sole task of transforming the multimedia content. It is fairly easy to maintain and effective in cost and adaptation to users and providers.

2.2 Streaming and adaptation mechanism

Recent years have seen tremendous increase in available multimedia content and diversity of portable devices. With content sharing sites and traditional TV services switching over to Internet as a main delivery channel, the amount of traffic generated by streaming (North America, Europe, Asia) is taking more then 50% share of the whole traffic [46]. The number of devices that can consume streaming multimedia content has increased and we see that the same phenomenon is happening with mobile internet traffic, i.e. more and more traffic share has been generated by video streaming with portable devices. Multimedia content providers need to address limitations of the devices and meet the users requirements by adapting the multimedia content.

These late developments have motivated much advancement and research in streaming and adaptation mechanisms. We examine the latest trends in streaming and adaptation in the rest of the section.

2.2.1 Streaming

Most of video streaming today is being delivered using HTTP [57, 33, 45] which leverage the existing web infrastructure using existing HTTP servers. HTTP offers a platform for providing video on demand (VOD) service because of its accessibility and openness. Traditional streaming protocols like Real Time Messaging Protocol (RTMP) can often experience problems with routers and firewalls, where packets gets blocked, while HTTP packets usually have no problem getting through.

HTTP is a stateless protocol meaning that there is no state management between between player and the multimedia server. The player needs to monitor the conditions and communicate with the multimedia server when the bandwidth changes. If the network bandwidth degrades, then the player signals to the multimedia server and starts to buffer the video long enough that it can seamlessly change to using a lower bitrate.

Video streaming services are mostly concerned about the video bitrate of the video delivered. The reason being, as stated previously, is that

network bandwidth controls the viewers ability to retrieve video and play it smoothly, and changing the video bitrate might in practice impact the buffer size and increase stuttering of video.

Popular multimedia providers as YouTube , Netflix and others, encode videos with several bitrates (usually denoted by it's quality) aimed for different bandwidth and resolutions. Videos are delivered with the either a method called progressive download or adaptive bitrate streaming (ABR). In order to efficiently use those methods, videos are made of small fragments which are continuously delivered to clients as long as the client asks for more fragments. The clients monitor bandwidth and CPU in the host, and if the context changes it sends an update to a server which opens a new connection and starts streaming media with the new bitrate.

Progressive download is an older delivery method where the video quality cannot be adapted once started streaming. The work progressive means that video does not have to be completely downloaded and can start playing as soon as some initial buffer is filled with data.

ABR is a modern delivery method where the video quality can be adapted while streaming. The server can change video size by sending a stream of video fragments with a different bitrate. The server depends on the client application to react to changes, e.g. CPU and network conditions which the server needs to change the video stream.

There are several industry leaders which provide ABR service and one that is becoming international standard.

1. Adobe Dynamic streaming for Flash, also called HDS
2. Apple HTTP Adaptive Streaming for iOS, also called HLS
3. Microsoft Smooth Streaming, also called HSS
4. Dynamic Adaptive streaming over HTTP, also called MPEG-DASH

YouTube is not an ABR service, at least not completely. Traditionally YouTube has delivered a progressive video download, where the file is stored on the hard drive, as opposed to the ABR streaming where file is only cached in the memory. A survey of YouTube streaming service [22] explains how the progressive download method works with the server controlling the traffic to the client. Lately there has been a shift to client controlled streaming where clients request the video data from the servers. The YouTube client application monitors the bandwidth and sends request to the YouTube server for a new stream. It uses a buffering mechanism to control streams so the user does not notice the quality adaptation while the new stream is buffering.

2.2.2 Adaptation mechanism

As seen in the previous subsection the majority of market leaders are adapting multimedia content prior to the streaming. This approach has a serious drawback: the multimedia content cannot be adapted to each available

device. The servers are wasting both disk and network resources by keeping many copies of the same file and sending a less optimal stream through the network. Still this a-priori transcoding method is widely used and to the best of our knowledge we could not find any large scale commercial streaming vendor that is dynamically adapting the multimedia content at runtime.

There are however numerous research projects ongoing in this field. Projects are categorized by scientist within different goals for the adaptation which either is multimedia content personalization, low energy consumption or QoS for mobile devices. We present two projects from each category. The research projects presented here are picked up because of the interesting approach they have on multimedia content adaptation. There are many more which are not included but are however mentioned in the survey papers [2, 29, 51] we used to find the relevant reading material.

Multimedia content personalization

Leopold et al. present a project called Component Based Multimedia Adaptation Framework [35]. The project has the goal to set up intelligent servers made of adaptation components which are dynamically integrated into adaptation process based on original encoding, target context and users preferences. They use meta-data for describing multimedia content, particularly MPEG-7 media information which stores data for codecs and storage formats. For describing host environment and user preferences they use meta-data from MPEG-21 Digital Item Adaptation (DIA) part of the standard. They identify three main categories of adaptation: Selection (select either video, audio or text), modality (transform multimedia content type to a different type, i.e video to image frames) and scaling/transcoding content. All of the named categories can have a set of adaptation components where the server needs to decide which adaptation to apply to given stream and meta data. The output of the adaptation process can be various versions of original file which suits the host environment and users needs. The result of the project shows that *"a component based software development approach and declarative behavior specifications are valuable means when developing extensible multimedia systems"* [35].

Tong et al. present a project "A novel model of adaptation decision-taking engine in multimedia adaptation" [52] which uses a Adaptation Decision Tree (ADT). ADT tree represent a sequence of adaptation decisions based on user preferences for modality and format. To use their explanation *"Before making decision, an optimal multimedia variation set (OMVS) with respect to user modality preferences is constructed and any element here is with the shortest distance to user format preferences for every modality. Therefore, adaptation decision can be executed by letting the element in OMVS travel along the ADT one by one."* The result of the project implementation and examinations has shown that ADT has better performance then other models such as Search Model (SM) and Overlapped Content value model

(OCV).

Low energy consumption

Moldovan and Muntean show in [40] that lower video bitrate will decrease battery use on the mobile devices by decreasing incoming packets count or sending packets in bursts. They explain that usually the multimedia clips have the same bitrate during the streaming, which is unnecessary for some clips that don't include much information. They present a mechanism called BitDetect which is used to detect good enough quality threshold of multimedia clips that users are satisfied watching. The result from their subjective study of BitDetect mechanism indicate *"that full-reference metrics in general and PSNR and SSIMmetrics in particular, can be used to detect good quality bitrate thresholds for multimedia clips with different content types and dynamicity, and for different video resolutions."*

Chen et al. propose a system called Anole in [10] used for energy-aware multimedia content adaptation. Their approach works only on the client device where they incorporate a framework which turns of background services while multimedia is consumed and adjust video bitrate, brightness and other device aspects. Adjustments are done at small levels so that the user is not experiencing a noticeable decrease in quality. The results show that by using this framework they can save up to 30% of the battery life.

QoS for mobile devices

Bellinzona and Vitali present a multimedia transcoding framework in [4] called Alembik. The framework uses a real time server side processing of multimedia. The device characteristics and application requirements are specified by tags which are defined in WURFL library. Multimedia content adaptation is done asynchronously.

Kim and Yoon [32] present an Universal Multimedia Access (UMA) adaptation system where they use MPEG-7 and MPEG-21 standards as descriptors for multimedia content adaptation. They use a server adaptation engine to process incoming media descriptors and state of the art adaptation tools in the adaptation algorithm. The system they designed looks very interesting but Kim point out that MPEG-7 and MPEG-21 standards requires licensing conditions so wider adoption of the system might be challenging in practice.

2.3 Summary

We have seen many advancements in the use of multimedia streaming and content adaptation. The multimedia content is usually adapted at the server side prior to actual streaming. Research projects try to advance streaming by introducing different approaches into solving dynamic adaptation.

The work in this thesis is inspired by the the novel method in the TRAMP project which is using middleware with context sensing and adaptation trigger mechanisms. These components are pre-installed on host machines and serve as a link between adaptation mechanisms and multimedia client applications. TRAMP middleware components aid both clients and servers to work separately without the need of knowing characteristics of the host environment in advance.

Chapter 3

Planning the project

This chapter describes the design and the architecture of the ATM library. We start with use cases and the general requirements description which are later used in the evaluation of the ATM.

3.1 ATM use cases

To illustrate a need for content adaptation triggered by the ATM we present two real use cases where the ATM automatically triggers content adaptation. These uses cases are referred to as *case 1* and *case 2* when we test with different context parameters.

1. A user is watching a stream video on a smartphone device connected to the a wireless access point. As the user moves away from wireless access point the smartphone switches to a GSM 3G network. The given use case demonstrates context change from a wireless network to a GSM network. The multimedia content needs to adapt depending on the network characteristics by changing the video bitrate.

2. A user is in the train watching a video on his smartphone. Suddenly many passengers arrive in the cabin and much noise is produced. A smartphone has a sound sensor that sense a noise level, and triggers event when sound goes over a certain threshold. The ATM listen to this event and then trigger content adaptation. Content adaptation in this case turns off the audio, i.e. send a message to multimedia server to stop sending audio stream and turn on closed captions.

Both use cases involve consuming media on a smartphone device. Due to the short time in this thesis we simulate the context change and test ATM on the PC.

3.2 Requirements

Table 3.1 shows the main requirements for the ATM. The background column shows the reason for the requirement and result column shows

Table 3.1: ATM requirements

Requirement	Background	Result
Support video streaming over internet	Type of application that TRAMP targets.	Show video from YouTube .
Decoupled from applications	Developers should not re-design or re-implement our adaptation mechanism. Multiple applications can using same solution.	Process run separately
Scalability	Support several applications simultaneously	Starting many players and report CPU, memory.
Small footprint	TRAMP targets resource-limited devices beside high-end devices	Space on the disk.
Low resource consumption	TRAMP target portable resource-limited devices beside high-end devices.	Amount of resource utilization (memory, CPU, bandwidth utilisation -packet sizes-).
Efficient	Consumers of multimedia application can tolerate up to 100 ms of delay.	Timestamps from file changed to sending request to a server.
Open and extendible	Other developers might add new features and extend the ATM.	Keep different ATM parts in different components with well defined interfaces

what is achieved by implementing the requirement and how is that achievement evaluated.

These requirements are used as the basis for the design and the implementation decisions. Most of the requirements are non-functional which imply how the ATM is evaluated while the one single functional requirement - to support streaming video over the Internet is fundamental purpose of the whole system.

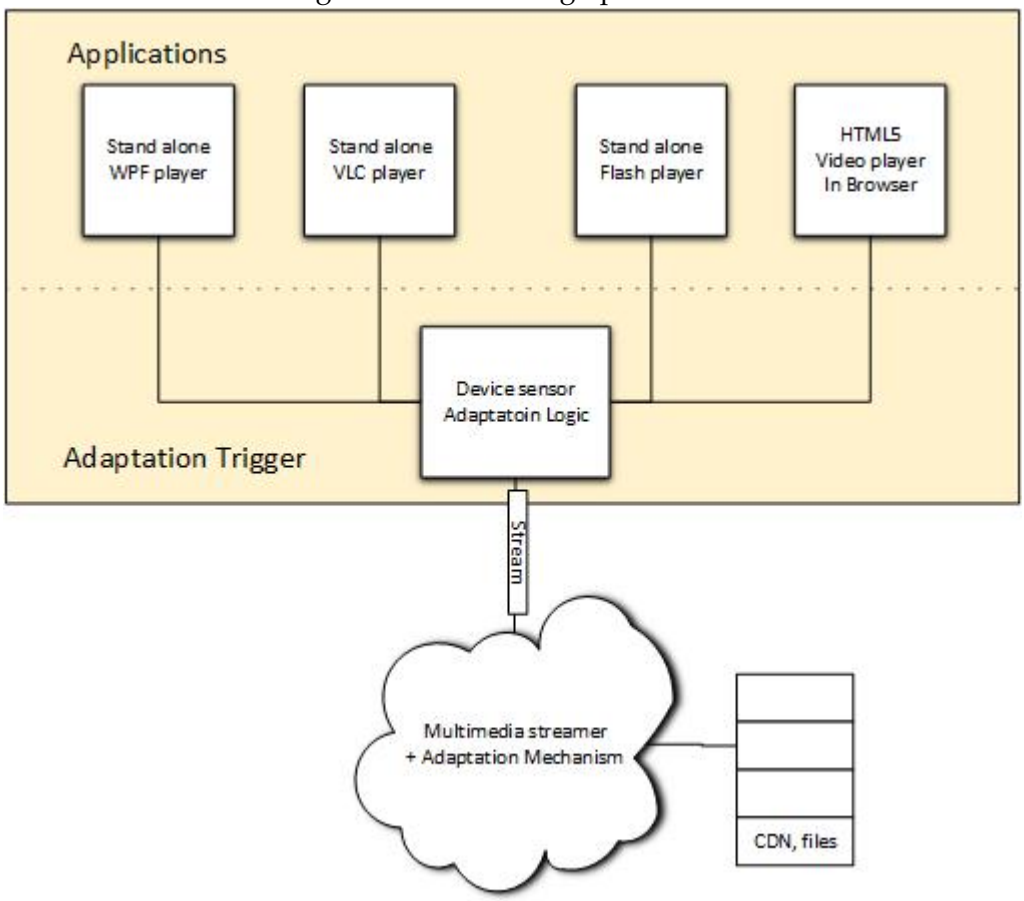
3.3 Design

We propose a solution of an ATM as a library loaded by a Windows console application where interaction to the client applications is abstracted away with Dependency Injection (DI) framework [54]. The proposed ATM has three main components:

1. Context listener component,
2. Context parameters processor component, and
3. Inter process communication component.

The components of ATM are separate libraries compiled in distinct files which are loaded into main application by the DI framework. There can be different implementations of the same component and DI loads only ones that are needed for present client applications.

Figure 3.1: Main design parts.



For the client applications we propose use of existing players with three components:

1. Video player component,
2. Code wrapper that calls video players API, and
3. Helper library that handles communication with ATM.

Figure 3.1 shows three main members of our design:

1. a client multimedia application with the code wrapper,
2. an ATM, and
3. a multimedia server.

The members we design are the code wrappers of client multimedia application, and the ATM.

The host platform is a Windows PC where all designed software is running. Windows OS was chosen for it's popularity and our familiarity with the environment. Recent research for usage share of operating systems

Table 3.2: Desktop operating system statistics on Net Applications

Operative system	Market share
Windows 7	48.77%
Windows XP	27.69%
Windows 8	11.30%
OS X.	7.58%
Windows Vista	2.99%
Linux	1.49%
Other	0.18%

done by Net Applications [16] shows that Windows is still predominant OS on desktop computers. Table 3.2 taken from Wikipedia document [55] which refers to Net Applications survey [16] shows current usage share of operative systems on desktop computers.

Other choices might as well be a Macintosh or a Linux machine, or essentially any other operative system capable of hosting a ATM, and running a client video application. Since portability issues are out of the scope in this thesis, we compile the ATM for Windows environment and suggest portability for a future work.

3.3.1 Client application

The client application that we use in the thesis need to be able to:

1. play video stream, e.g. YouTube stream, and
2. expose an API so that we can control it through a code wrapper.

We take a look at existing video players that are popular on the Windows platform. We investigate video players that support playing of network video streams and most specifically can render YouTube streaming multimedia content. The non functional requirement is that the player can be manipulated with a custom code. We design an implement code wrapper for controlling the video players through exposed API. The code wrapper is also needed for handling the communication with the ATM.

One of the main requirements for the ATM is to work with multiple and different client applications at the same time. To that end we want to use several video players and test the content adaptation with all of them. There are two main types of players, standalone application and browser (video) plugin. We develop code wrappers and test with both types.

Players used in the thesis are:

- Standalone players:
  - LibVLC wrapped in .NET container,

  - Flash video player wrapped in Adobe Shockwave Flash container,
  - MediaElement from .NET Windows Presentation Foundation (WPF) library, and
  - Windows 8 Store Video app.

- Browser players:
  - Flash player object - plugin,
  - HTML5 video element, and
  - Silverlight video player - plugin.

Standalone players

LibVLC [36] is a audio/video library developed by VideoLan under GNU License. It is a widely used media library and has long list of features. It is compatible with various video formats, including the H.264/AAC, and supports several video outputs like OpenGl, Xvideo, DirectX and others. It is also portable with wide range of operational systems. LibVLC is used as a media component for VLC player itself and can do everything VLC player does. LibVLC core is written in the C language.

Flash video player [1] is a Flash component that can be loaded into Adobe Shockwave Flash container and controlled through the code. It is a proprietary software owned by Adobe and is often called web standard for multimedia playback, widely supported and used in practice. It is distributed as a plugin which a user has to install for a specific browser and can be used for playing FLV streams. The Adobe Shockwave component work as a container that loads Flash video player. It can be accessed through .NET code and modified to open YouTube videos and call the YouTube API.

WPF MediaElement [61] is a standard .NET video/audio component. It can as previous two players render a H.264 stream, but has otherwise limited codec support and functionality, with standard out-of-the-box support for wmv and wma codecs. It can play a YouTube stream, but only within MP4 container. We include this player to test the standard .NET functionality.

Windows 8 Store Video app [59] is a new type of application that runs on Windows 8 and newer platforms. We have included it in the test to be sure that the ATM is future proof. The test has shown that this type of apps works as a sandbox with no direct way to control it as with previous players. The application itself can reach out and use web services, but is otherwise closed to direct communication.

Table 3.3: Container formats

Container format	Video codec	Audio codec
MP4	H.264	AAC
WebM	VP8	Vorbis
Ogg	Theora	Vorbis

Browser players

There are two types of browsers players, a plugin player, which is a software component that adds new features to existing browser application, and a HTML5 video element which is implemented by every browser in a special way and aims to replace plugin players. They are both important and, as we see below, the most multimedia content providers are targeting especially those two player types.

Flash player object is a plugin that users have to download and install in the browser. Once installed it can play Flash video which is supported by many popular vendors. Flash player has been in use since 1996 and has became a predominant choice of delivering video content over the internet. Flash is almost ubiquitous software and 95% of PCs have Flash installed. It uses FLV and F4V container file formats for delivering video and audio. Flash video had to face a challenge of supporting diversity of smartphones. As a result Adobe has in 2011 announced that they decided not to extend support for making Flash work on mobile devices.

HTML5 video element is a standard way of showing videos in the browsers [25]. Before the HTML5 video element there was no standard way of video rendering and browsers were depending on the plugin, such as a Flash player, to render a video stream. HTML5 video is essentially an html <video/> tag which every browser implements in its own way. It is an open source and a lightweight player that is becoming more and more popular in use. There are currently 3 supported video formats for HTML video element which are shown in the Table 3.3 taken from [27].

Silverlight video player The Silverlight player is the Microsoft answer to Adobe Flash player. While not as popular as Flash, it still is a popular tool to deliver streaming video for some of the major multimedia content providers, such as Netflix. We are not going to go into details about strengths and weaknesses of Silverlight as the latest trends in Microsoft suggest that it might not be supported product in the future. It still is worth to mention that Silverlight player offers live adaptation streaming with Microsoft HSS service and is worthy competitor of the other two players.

From the presented standalone players we use LibVLC, Flash video player and WPF MediaElement in testing with the ATM. From presented browser players we use YouTube iframe player described more in Subsection 4.1.1, which can be either Flash or HTML5 player depending on the browser. The

rest of the mentioned players, the Windows 8 Store Video App, HTML5 video tag and Silverlight app were considered and tested but we found out that they were infeasible to continue with because of the limitations we found described more in Chapter 4.

For each of the chosen players we develop a special code wrapper. The wrappers have a tight coupling with the video player controls, and include an external helper library (hereby called video-helper-lib) for handling communication with the ATM.

3.3.2 ATM

The two use cases described in Section 3.1 illustrate the need for a content adaptation. In *case 1* the device's environment is changed from a wireless to a GSM and available bandwidth is potentially different, while in *case 2* the user context is changed by the change in user's environment. Some client application cannot react to context changes itself and something else is needed. The ATM takes up the role of aiding those client applications in reacting to context changes, handling the communication to the multimedia server which both streams and adapts the content, and sending the stream to the client after it has been adapted. The context that ATM reacts upon is a device and user environment characteristics and described in more detail in subsection 3.3.3.

The ATM has three main responsibilities:

1. listening to changes of user and process context,
2. collect new context data, process the data and prepare the API call to the multimedia server, and
3. communicate with the multimedia server and hand over the adapted stream to the client application.

The ATM is designed as a library loaded and linked by a another software component, such as console application or a service (daemon). When the library is loaded it starts providing a service of sensing the context and sending the changes to the adaptation mechanism. It creates a set of context parameters based on collected context data and either updates the video stream url by adjusting the Media resource locator (MRL) or notifies the video application with a new event sending the context data.

One of the main requirements for the ATM is to work with multiple client applications, and also support different type of communication, such as named pipes, TCP sockets or shared memory. In addition we want ATM to react on different type of context change, which can be issued by device drivers such as a graphic card driver or an audio card driver, device sensors such as a GPS sensor or a noise level sensor, and user preferences for given context or combination of contexts.

Because of these requirements we design and decouple three main responsibilities into different libraries and use the Dependency Injection

(DI) framework to load the ones that are needed for the current context. Dependency Injection is a pattern described by Martin Fowler in his document [20]. The main reason for using the pattern is to free the classes of responsibility for instantiating their dependencies. The dependencies are in this case the interface implementations for adaptation parameters calculation and handler for communication. They are "injected" in the setter part of the classes from by a DI framework that we describe in the Implementation chapter.

Type of application loading the ATM

We investigate two options for the main application that loads the ATM, a Windows service (daemon) application and a Windows console application. There are few advantages and disadvantages with both alternatives:

Windows service

- Pros
  - Run in the background
  - No need for user to login, can run before login happens
  - Automatic start on Windows
  - Failure policy defined
  - Can have elevated rights
- Cons
  - No option for UI or direct interaction
  - Do not share memory space with user application and session

Windows background application

- Pros
  - Can have a console UI.
  - Can share memory space with other user applications — this is really important for our design.
- Cons
  - It runs in a user session, i.e. after user has logged on.
  - No out-of-the-box automatic start up and failure policy, but can be implemented.

We chose to load ATM with a Windows console application because of the shared memory consideration which is described in the implementation part.

Inter-process communication (IPC)

The ATM and the client application with code wrapper are two processes that need to use inter-process communication in both directions. We consider inter-process communication options supported by Windows. The Msdn website [28] explains the IPC options when using a Windows platform.

The types that we have considered to use are:

- Windows Sockets, and
- Named Pipes.

The other IPC types are useful in different cases, such as RPC or shared memory. The Windows socket and the named pipes are the ones we found the most useful for using with video streams.

Both Windows sockets and Named pipes are good choices for our IPC needs. They both provide asynchronous calls between the video player and the ATM. The communication can either be one way or a duplex, which is the one we use. They are both comparable with regard to performance, especially if the process are on same machine or in the same local area network environment (LAN).

Windows socket transmissions are more streamlined and have less overhead because they take advantage of existing TCP performance. Named pipes have little or no overhead for adding new clients which use the same pipe, and up to the order of hundreds can be simultaneously connected. The number of pipe instances is limited by available non-paged pool which is physical memory used by the OS kernel. We can specify MaxConnection property on the Named pipe biding which gets or sets number of allowed connection.

It is important to note that since communication part is abstracted away then the ATM can actually use whatever implementation for communication handler is loaded by the DI framework. If the client application needs to use a socket then the socket class is instantiated and added to communication handlers collection.

3.3.3 Context change sensing

In Section 2.1 we used Days explanation of context [17]. He define context to be any information used to characterize the situation of a person, place or an object. The context can be used to describe many different conditions of environments, situations, preferences etc. This is a very wide term and for the purpose of this thesis we narrow our meaning of the context to describe the host device and the environment the device happens to be in. Host devices have a number of important context characteristic such as display, which can have different size, resolution, color depth, have touch screen or not, etc. The host can have a radio receiver, a GPS receiver, a Wi-Fi receiver, a Bluetooth and an Infra-red receiver and others. It can have many sensors such as gyroscope, noise sensor, temperature sensor, camera sensors and

others. All this is used to describe the context of a host. Environment context is used to describe network conditions, noise conditions, time of the day, place and other situations.

We have already stated in Section 1.5 that the real context changes are simulated by reading configuration parameters from the file. From described scenarios in Section 3.1 the parameters need to include the video bitrate used when network bandwidth changes, and option for including or excluding the audio stream when device environment has a high noise level.

We intend to describe the adaptation parameters in a way that is not bound to any specific language or OS. The predominant approach for describing environment characteristics in by using XML based Resource Description Framework (RDF) [14]. We use similar XML notation to create adaptation descriptors.

The Listing 3.1 shows the file used for describing the needed context. We are focusing on two parameters, which are important for the use cases presented in Section 3.1. They are <supportedQuality>, which is an option for providing the YouTube with a bitrate quality, and in the cases where there is too much noise we are using <audio> with mute option to trigger no audio and closed captioning turn on.

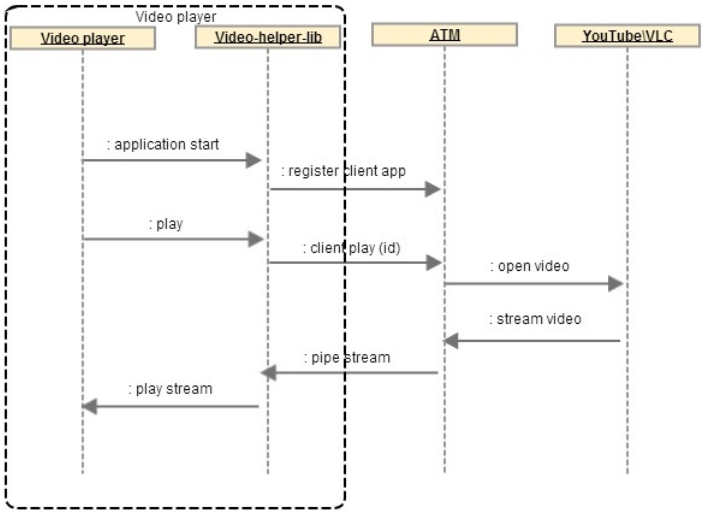
These are just examples of the possible settings. In Section 2.1 we describe many different adaptation options which include many categories. The presented options in the Listing 3.1 are used when context is adapting according to the target context, i.e. network and user context, i.e. user environment. We also expect adaptation to be dynamic, i.e. at the time requested.

Listing 3.1: Context parameters in XML

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<settings machineName="Dell Latitude E5350" operativeSystem=
    "Windows 8.1 Edition">
  <process_context>
       <supportedQuality>tiny</supportedQuality>
       <audio>mute</audio>
       <captions>on</captions>
  </process_context>
</settings>
```

To listen to file changes we use *FileSystemWatcher* class from .NET library which raises an event every time the watched document is edited. The trigger event starts the adaptation update process which collects new context data and send the data to whatever class is implementing the communication interface, which is a *NamedPipeServer* in our case. The data are packed in an object called *CurrentContext* which has a string value for *VideoQuality* for *AudioQuality* and a bool value for *Captions*.

Figure 3.2: Initial design - video player initiates stream.



One important detail is that the ATM has the responsibility to convert context data, i.e XML settings, into a method calls that exists in multimedia server API. We assume that this can be done within the ATM logic and test with implementing how multimedia server API can be called.

3.3.4 Multimedia servers

In this thesis the multimedia content provider for the client video application is a legacy multimedia server and we use YouTube as the main source for video streams.

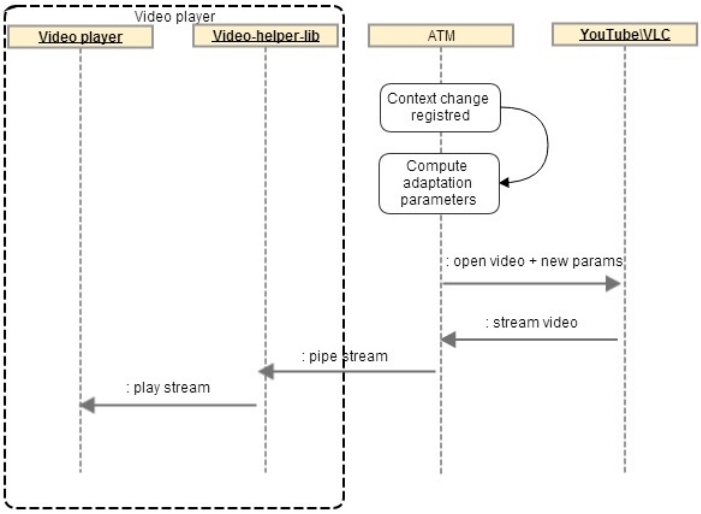
YouTube streaming service is one of many possible legacy streaming services, such as Vimeo, Hulu, Metacafe or NRK TV. The real strength of YouTube is an excellent community support, developer tools, programming language support and documentation. It is today the most used sharing service and the most of the multimedia content is created by individuals for free.

In addition to YouTube , we use a VLC command line to try to transcode a YouTube stream into different codecs and filter out video and audio when not necessary. VLC is one of the market leaders when it comes to available features for transcoding a multimedia streams.

3.4 Interaction scenarios

There are two common scenarios where we see the video player and the ATM start interaction. The first one is a shown in a sequence diagram on Figure 3.2. Here is the video request initiated by the player. Upon application start-up, the video player calls a video-helper-lib to initiate a handshake with the ATM thus registering it for future context change updates. After handshake succeeded video player sends a command play to the video-helper-lib which in turn handle communication with ATM.

Figure 3.3: Initial design - context change initiates adaptation.



ATM contacts the multimedia server and issue a video stream request which is sent back to the video-helper-lib and further to the player.

When the context change happens, it is up to the ATM to react and initiate the video adaptation process which results in the new video stream. This is shown in the figure 3.3. Here, the ATM processes a context parameters file written in XML with parameters needed for the adaptation, as explained in the Subsection 3.3.3. It sends the context parameters to the multimedia server and handles the adapted stream back to the video player.

Chapter 4

Developing the project

In this chapter we describe implementation of the video player wrappers, the ATM and their interaction. We finish with describing the challenges discovered and the continue researching with simulated components.

4.1 Implementation

For the development framework of the client application and the ATM we considered two alternatives, .NET and Java.

Both frameworks are a good choice for developing the software in this thesis. They have many similarities such as both aim to simplify the development and let the developers focus on the business logic, while the framework provides more fundamental services. Both frameworks provide components for standard way of performing tasks related to databases access, web pages scripting, message handling and connecting to remote sources and services. There are also many differences in inner workings of frameworks, and other aspects of the languages.

For this thesis we choose to use the .NET framework for several reasons. .NET framework is a free product by Microsoft which is highly extensible and has many third-part additions available. It is a very good tool for targeting Windows platform and also supports connectivity with non-Windows systems.

The output of the .NET compiler is a non-executable-code which is executed by the Common Language Runtime (CLR). The CLR converts the compiled code into machine instructions which are then executed by the computer's CPU. The Mono project has made .NET available on Linux and MAC so that applications are not bound to CLR and can use Mono runtime to execute on the those platforms as well.

.NET code can be cross-compiled with tools like Xamarin Studio [62] so that the main code base is kept equal for every targeted platform. Having two code bases for the same application is not considered a good programming practice.

The main reason for using .NET is our familiarity and technical skills with the framework. We use streaming tools from Windows Communication Foundation (WCF) [37, 7] which is a .NET framework for building

service-oriented applications [60]. Acquiring Java framework skills in the short time given for this thesis turned out to be timewise infeasible and non practical.

There is one exception in usage of .NET framework when we use Java Spring framework to test websocket implementation of the HTML5 and YouTube iframe player. Rest of the thesis development is done using .NET 4.5 framework version and running on Windows 8.1 Edition OS.

4.1.1 Client applications

In this section we describe inner workings of five video player wrappers:

- HTML5 video element and later YouTube iframe player
- WPF mediaElement
- Windows Form wrapping Adobe Shockwave object
- WPF wrapping libVLC component
- Windows 8 Store app and Silverlight app

HTML5 and YouTube iframe player

The first video player wrapper we implement is a wrapper for a HTML5 video element. HTML5 video element is essentially a HTML <video> tag which resides in a web page. It currently supports three video container formats, which are MP4, WebM and Ogg. It is new in HTML5 definition, and not supported by older browsers.

HTML5 video element looks like the Listing 4.1 as described by W3C at their video wiki page [26].

Listing 4.1: HTML5 video tag

```
<video width="320" height="240" controls>
  <source src="http://domain/movie.mp4" type="video/mp4">
  <source src="http://domain/movie.ogg" type="video/ogg">
  <source src="http://domain/movie.webm" type="video/webm">
</video>
```

This is all that is needed to start using a HTML5 video element in the player wrapper. The presented video tag includes three source tags with different url for different containers. This is because every browser implements the video tag differently and use a different video container. As an example a Firefox browser, a product of Mozilla company, promise never to add support for proprietary codecs. They currently support Ogg container, and according to latest Web browser market share trends, provided by W3Counter [38], Firefox has 18.1% penetration, which is a large share of the audience.

HTML5 video has a great challenge of standardizing a video player and codecs supported. HTML5 video is not able to play YouTube stream out of the box, because providing it with a YouTube url does not give a direct link to the YouTube video file.

We investigate next how YouTube has implemented video player to support many browsers. YouTube uses essentially both Flash and HTML5 players. Flash is used with browsers that have installed Flash plugin, and HTML5 is used otherwise.

The initiate the player in the web page by using an iframe, an the usage is pretty simple and presented in the Listing 4.2.

Listing 4.2: YouTube Iframe usage

```
<iframe id="ytplayer" type="text/html"
src="https://www.youtube.com/embed/videoID&enablejsapi=1" />
```

The <iframe> tag or inline frame is a HTML tag which is used to embed another HTML document within the current web page. It is used to insert content from other sources and YouTube is using it to insert the YouTube player into the parent document.

When a user opens a page that contains an YouTube iframe, a GET request is issued in the background to the main YouTube site [3, 19]. The server reads a HTTP header from the GET request and loads either a Flash or HTML5 player. It also provides a direct link for the video source attribute from a different server location (usually CDN) which is decided at runtime.

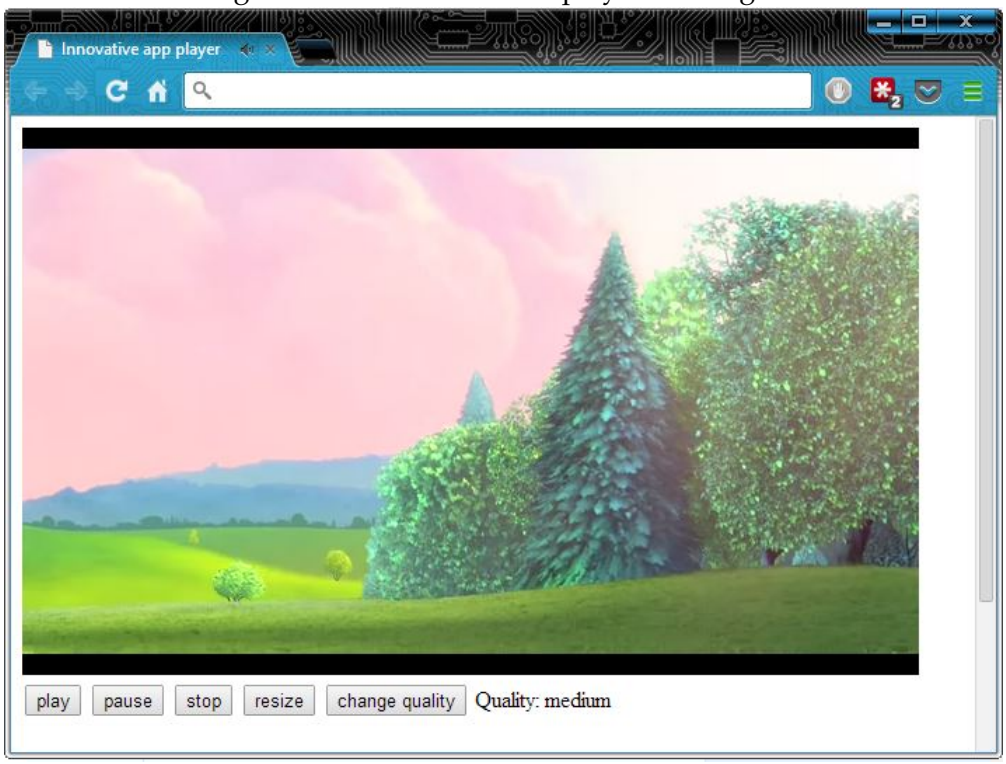
YouTube players (both types) are controlled either by clicking on certain areas on the video player (playback buttons) or by using a YouTube player iframe API [64]. Iframe API is essentially a set of javascript methods and events which are defined for the iframe player.

The player wrapper that we develop uses custom controls for the video playback. The player is shown on Figure 4.1. It is a part of a web page which uses HTTP request-response protocol where the client sends a request and the server sends a response to the previous request. This is because HTTP uses a TCP connection which is closed after the response-request phase has finished, and the client needs to reinitialize a new TCP connection next time it contacts the server. Without this connection the server is not able to contact the client.

When the server wants to send a message to the player it has to work around this stateless nature of HTTP. There are three ways the server can achieve this. One way is to use continuous polling to the server with a defined frequency. This is not a good option for our case scenario. Either the polling frequency is high and we keep polling the server unnecessary, or the frequency is low and unacceptable for users who wait for a multimedia content to adapt. Other two options, Comet and Websocket, can be used and we describe them next.

Comet [13] is an older model of using the HTTP protocol to simulate a server-push request. It works in two steps. First, the client issues a request

Figure 4.1: YouTube iframe player running



to the server and the server holds this TCP connection open until there is data to send. Second, when the server responds with the data, the client receives it and issues a new comet request. This approach has been tested and works in practice, but introduces few challenges. One is that keeping open connection on the server requires much of resources. Other is that time-outs are bound to happen and handling them is not a trivial issue in some server environments.

The third option, which we find efficient and use in the implementation, is a Websocket [58]. Websocket is a new technology which essentially uses TCP to create a duplex connection allowing client and server to send messages simultaneously. The TCP connection is still kept open as with the Comet model, but the server does not need to use a thread per client, and can act asynchronously. Websockets do not use HTTP except for the initial handshake when the HTTP servers interpret the handshake like a HTTP upgrade request to a websocket. All the latest web browsers are currently supporting the latest websocket specification [58].

To use a websocket functionality we implement a JavaScript methods to open a socket, send a message with a socket and subscribe to event called "onmessage". Onmessage event is defined in the websocket API and is triggered when websocket receives a message. The event is also used by the web server which in turn uses the ATM. When calling socket open we need to specify a url to the websocket server. We develop a websocket server as a simple Console application. The server just calls start, waits

for connection, and when client initializes a connection request, the server stores it into a websocket collection. Upon sending the server loops through collection and send message to each item by calling websocket API method "sendMessage".

Content adaptation We test content adaptation for both use cases. For the use *case 1* we only need to adapt a bitrate, which in YouTube player means adjusting a video quality. This is done by calling a method *player.changeQuality('quality-string')* For the use *case 2* the YouTube player needs to turn off the sound (most desirable is to completely exclude audio streaming) and switch on using closed captions. The audio can be muted by calling the JavaScript method *player.mute()*, however the YouTube iframe API currently does not have a method for turning closed captions on and off.

The logic for adapting content for both use cases is included in the player code wrapper. The wrapper reacts on a event triggered by the video-helper-lib when it receives a message from the ATM and call the required method.

Limitations *"YouTube videos are available in a range of quality levels. The former names of standard quality (SQ), high quality (HQ) and high definition (HD) have been replaced by numerical values representing the vertical resolution of the video."* [63]

This citing describes which adaptation options YouTube provides. We can only change a quality of the video if the content adaptation requires to do so. Our initial assumption that YouTube offers a great variety of adaptation functions has been proven wrong. Looking further to other multimedia content provides, such as Vimeo, MetaCafe, Dailymotion, NRK and many others, we found out that all of them are essentially doing the same as YouTube, changing the quality when the player sense that bandwidth and CPU usage change.

Another limitation of using a YouTube player is that it itself opens a connection to the media source so the ATM is not able to control it. The ATM can only send a message describing a context change and the player wrapper has to react to that change either by providing a new quality level (*case 1*), or mute and show closed captions (*case 2*).

A third limitation is that YouTube has not included an option for controlling closed captions setting in their API. User can click on that option in the player UI but not call the method from the code.

HTML5 player with VLC YouTube has limited adaptation options and we tried testing HTML5 player with a VLC streamer. VLC streamer is a part of VLC software which has many functions as playback, transcoding, streaming and can be controlled through an HTTP interface or a command line. It can stream a YouTube content with runtime transcoding.

We found some limitations here as well, which made us to continue testing with YouTube in different players. The stream that VLC produces

can be seen in a different VLC player. However using an HTML5 player the only container format we could use is Ogg. The quality of video was very poor and the video is not supported in every browser. Another limitation is that changing the quality, i.e. calling a new transcoding command while a video is running would only make the video start from beginning, or produce long adaptation delay which is not wanted behaviour.

Java framework used The web page including the YouTube iframe player together with the websocket server are the only software developed with Java Spring framework. We experimented with content adaptation by manually triggering onmessage calls from the websocket server which has a Java console application interface. The websocket server is of a glassfish server type. .NET has an equivalent websocket server called Microsoft.Websockets which can be used. Since we already founded adaptation limitation in the YouTube iframe player, we have not implemented communication with the glassfish websocket server and ATM. The code wrapping the YouTube iframe player would be the same regardless of the websocket server type, and result would be the same.

WPF MediaElement

MediaElement is a class in WPF System.Windows.Controls namespace [61]. It is a standard multimedia component used in Windows applications developed in .NET. It supports a limited number of codecs and video formats, but is capable of decoding H.264 codec in MP4 video format and can render YouTube stream with some special url adaptation.

Since MediaElement is a native component in the .NET library it is very easy to use in practice. One needs to reference it, instantiate a new instance and provide it with the video source link. What we need to figure out, is to how to provide a direct link to YouTube video that has a MP4 container format.

This can be done manually or by using an existing library for getting a direct video link to the YouTube video. We use a library called MyToolkit [41] which includes a method for getting direct link to the YouTube video and wraps it in a short procedure call. Procedure can be called as in Listing 4.3.

Listing 4.3: Using MyToolkit Youtube procedure

```
var url = await YouTube.GetVideoUriAsync("YE7VzlLtp-4",
    YouTubeQuality.Quality720P);
player.Source = url.Uri;
player.Play();
```

The process of getting the direct URL, which the MyToolkit is also using is described next. Usually a link to a YouTube video URL is specified by

video ID:

$$http : \backslash\backslash www.youtube.com\backslash watch?v = [VIDEO - ID].$$

When the request with the url above is issued, a YouTube server replies with a JavaScript data block which includes direct links to all versions of the video in all available qualities. Each video link is denoted by an ***itag*** [3] which is an undocumented YouTube video parameter. It is a numeric parameter which can be used to exactly identify the video container and quality. Parameter itag is used to find links to MP4 streams that MediaElement can render.

Listing 4.4: YouTube url with itag response

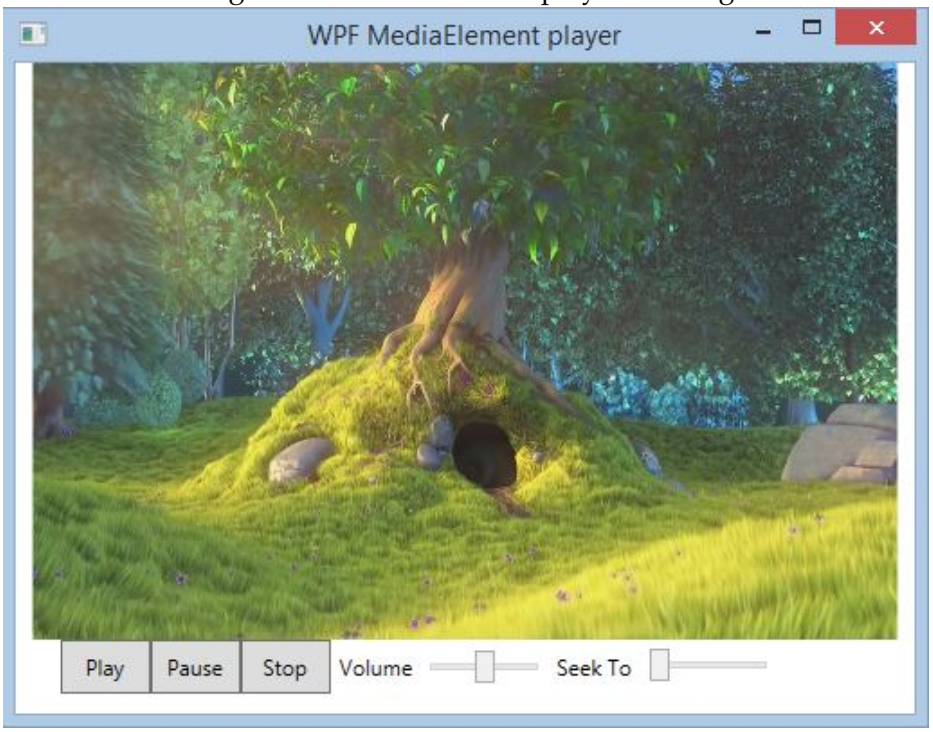
```
(1) "http://r2---sn-bxuovgf5t-vnal.googlevideo.com/
    videoplayback?key=yt5&ip=84.215.225.220&ipbits=0&itag
    =22&.."
(2) "http://r2---sn-bxuovgf5t-vnal.googlevideo.com/
    videoplayback?key=yt5&ip=84.215.225.220&ipbits=0&itag
    =43&.."
(3) "http://r2---sn-bxuovgf5t-vnal.googlevideo.com/
    videoplayback?key=yt5&ip=84.215.225.220&ipbits=0& itag
    =18&.."
(4) "http://r2---sn-bxuovgf5t-vnal.googlevideo.com/
    videoplayback?key=yt5&ip=84.215.225.220&ipbits=0&itag
    =5&.."
(5) "http://r2---sn-bxuovgf5t-vnal.googlevideo.com/
    videoplayback?key=yt5&ip=84.215.225.220&ipbits=0&itag
    =36&.."
(6) "http://r2---sn-bxuovgf5t-vnal.googlevideo.com/
    videoplayback?key=yt5&ip=84.215.225.220&ipbits=0&itag
    =17&.."
```

Listing 4.4 gives examples of the data block with different itags when requesting video with ID "YE7VzlLtp-4". Itag is the last parameter in each link. Itags 18 shown with index (3) and 22 shown with index (1) are in MP4 file formats. Figure 4.2 shows a running WPD MediaElement player.

Content adaptation Testing adaptation for *case 1* is simple matter of changing the quality option by providing a MediaElement with a direct link for MP4 file having new quality. *Case 2* is only partially done by muting a video. We were not able to trigger a closed captions since it is not included in the YouTube API.

Limitations The WPF MediaElement have similar limitations as the YouTube iframe player in regards to using it with ATM. Since it is

Figure 4.2: MediaElement player running



using YouTube as multimedia content provider then the only available adaptation option is the video quality. In addition, when it changes quality it starts streaming from the beginning. The closed captions are not part of the MediaElement API and we were not able to use it. The logic for content adaptation has to be within the player wrapper code.

Windows Form wrapping Adobe Shockwave object

We look next at Adobe Shockwave object wrapped in the Windows Forms application. *AxShockwaveFlashObjects* is a COM library which allows .NET developers show Flash content or play Flash video players. It is proprietary product owned by Adobe and is distributed as a part of Flash plugin instalment.

The AxShockwaveFlashObjects can help us to overcome the need for a websocket server that HTML5 player uses as explained in Subsection 4.1.1. It loads a YouTube Flash player (further called YTplayer) just by providing a regular YouTube URL link which we normally use in browsers and starts to play video. The YTplayer look exactly the same as it does in the browser and has the same control options presented in Figure 4.3.

Controlling the YTplayer through a code method calls is not a straightforward process. AxShockwaveFlashObjects itself provide a general API, but since we want to use a loaded YouTube Flash player we have to call a YouTube API in order to get the right response, which AxShockwaveFlashObjects does not know about. This means that calling regular YouTube API does not work and something more is needed to establish interaction.

Figure 4.3: Windows Forms Flash player running

There is one event handler and a method in AxShockwaveFlashObjects API that we can use for interaction with Flash and C#. YTplayer has event *FlashCall* which triggers every time YTplayer wants to signal an update and for every event described in YouTube iframe API [64]. The event argument is a XML response which we need to parse to extract the event id and the arguments. We use this event to start logging when YTplayer is ready and to track state change when it happens.

The method we call on YTplayer is CallFunction(flashXMLrequest) which takes a Flash XML as an argument. A sample XML to load a video looks like in Listing 4.5.

Listing 4.5: YTplayer XML request

```
<invoke name="loadVideoById" returntype="xml">
        <arguments>
                <string>YE7VzlLtp-4</string>
        </arguments>
</invoke>
```

The XML request is constructed at runtime every time we call a YTplayer YouTube API in order to control it or to retrieve information.

The YTplayer support only YouTube content delivered in FLV video

container.

Content adaptation Since this player is essentially the same as a Flash version of YouTube iframe player it can do exactly the same with regards to content adaptation. Both use cases can be done in the same manner.

Limitations The same limitations as for YouTube iframe player apply here. We have removed the need for a websocket server by calling YouTube JavaScript API directly from the C# code.

WPF wrapping libVLC component

LibVLC is an open source C library with bindings to many different languages and platforms, Windows and .NET included. As stated in [36] *"The **libVLC** (VLC SDK) media framework can be embedded into an application to get multimedia capabilities"*, which we describe next.

To include libVLC in .NET application, the wrapper has to map many C library calls to methods that .NET runtime understands. Since its not a native .NET COM component, the wrapping code need to extensively implement many reflection calls, memory management, pointer marshalling and custom classes and structs. We use therefore one of existing libVLC wrappers that can be found on the Codeplex website, specifically VideoLan DotNET library [56] which has been used and tested thorough in real projects.

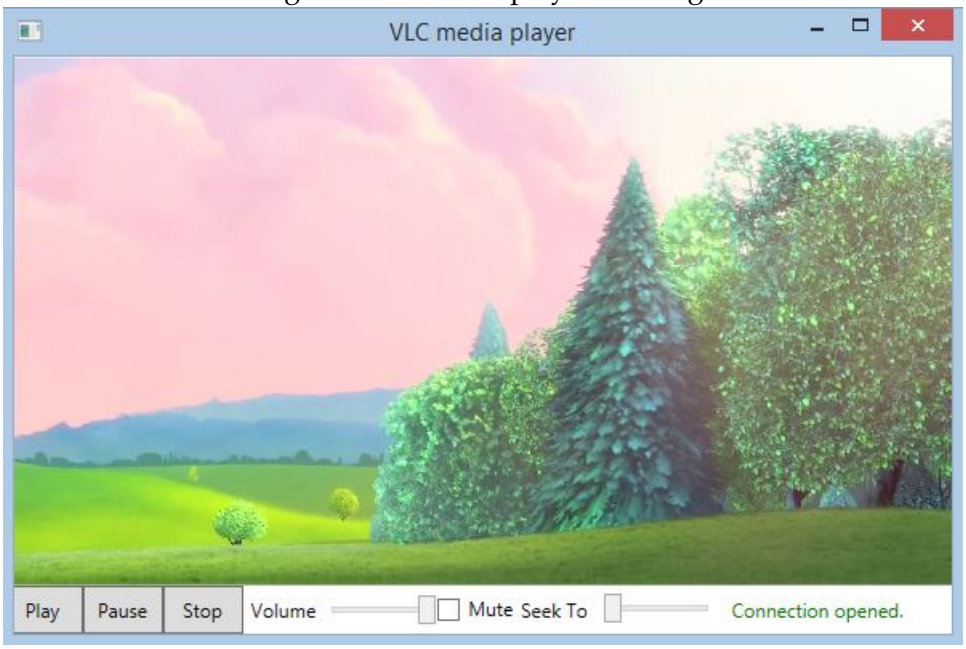
LibVLC has many options of controlling a video, many more then the other players. We appreciate a SetSubtitle option which adds closed captions to playing video. Not every video has added captions and getting them from YouTube requires parsing the response and creating a request in form of:
http://www.youtube.com/api/timedtext?v=VIDEO-ID&lang=code.
While writing this thesis this option was however removed from the YouTube API and we had to manually download the subtitle to disk and load it from there.

Link to a video is provided by using a Media Resource Locator (MRL) to a YouTube video. After the video is loaded we can control it with simple playback commands. The running player is shown in Figure 4.4.

Content adaptation The libVLC player can partially work for *case 1*. Same as WPF MediaElement libVLC can load a YouTube video with new quality by using a different link to the YouTube file with other quality. YouTube previously had an option to add quality parameters in url of the video together with the position. It still has position parameter, but the quality parameter has been removed completely. For *case 2* libVLC can mute the audio and show subtitle by loading a file from the captions link.

Limitations As with the previous players the limitations are the same regarding adaptation logic. LibVLC does not expose a media source link ob-

Figure 4.4: LibVLC player running



ject. It has an object which is a read-only property called **media** responsible for linking with the media source and controlling the streaming. The object is however not assignable and is controlled by the inner workings of LibVLC, which essentially means that the wrapper is not able to add it from outside of the application and the logic for reacting to context has to stay within the player component.

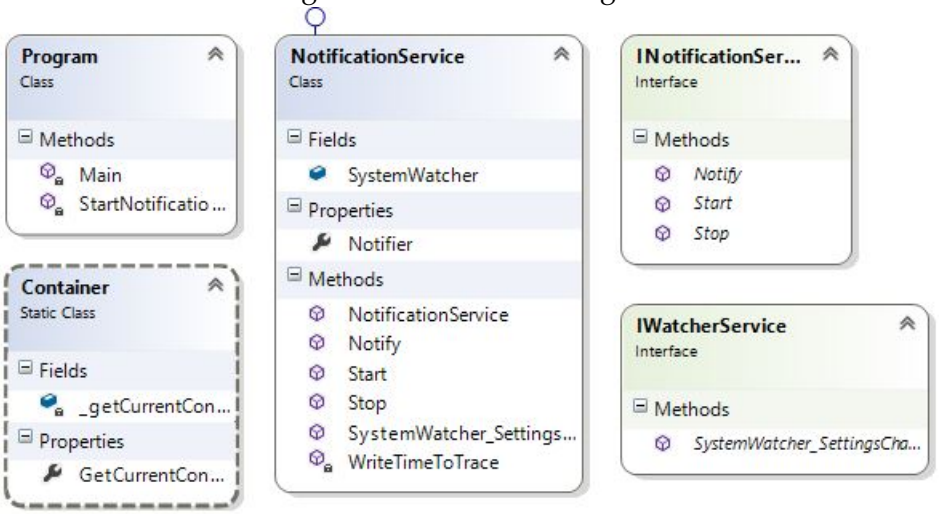
Windows 8 Store app

We include this player in our tests to check if the content adaptation is possible with Microsoft Smooth Streaming protocol (HSS), which the Windows 8 Store player is using. The player wrapper code is developed with XAML language. The video is streamed from a Microsoft HSS server provider.

Our initial test shows that here too the observable adaptation is very much the same as with the YouTube. HSS uses adaptive bitrate streaming method to deliver different quality of video to changing player environment. It essentially resembles the YouTube way of pre transcoding videos but the delivery method is different. Other adaptation options are lacking.

Another observation is that the player works in a sandbox, and cannot be reached through code directly. It can start connection from within, but is otherwise closed to outside interactions by design.

Figure 4.5: ATM class diagram



4.1.2 ATM

We describe the ATM requirements in the Section 3.2 which are referenced thorough the implementation process.

Regarding **Scalability**, **Small footprint**, **Low resource consumption**, and **Open and extendible** requirements, we use a set of collective implementation decisions. The DI loads libraries for context sensing and communication handling, which satisfy open and extendible requirement. The usage of the DI framework meets also the small footprint requirement by loading only needed libraries. The use of NamedPipe server with multi threading message dispatcher satisfies ATM requirement for the scalability.

Seeing that the player components do not have an open API for controlling the media link, the requirement **Efficient** needs to be implemented within the players wrapper logic.

ATM parts

The ATM is divided into three main parts:

- context listener,
- context parameters processor, and
- connection handler.

All parts are build into separate DLL files and tied with Microsoft Unity Dependency Injection framework. Unity DI uses configuration file to resolve dependencies.

A class diagram of the ATM is presented in Figure 4.5. The entry point of the ATM is the Main method in the Program class which calls StartNotifications. Container is a static class that initializes Unity DI container. Notification service is the main class which gets all dependencies

instantiated by Unity container. System watcher is instantiated as a FileSystemWatcher, while Notifier is instantiated as NamedPipeNotifier [43]. The configuration of DI is shown in Listing 4.6

Listing 4.6: Unity DI configuration

```
<unity xmlns="http://schemas.microsoft.com/practices/2010/
    unity">
    <container>
      <register type="NotifyDefinitions.INotifier,
          NotifyDefinitions"
                mapTo="NamedPipeNotifyImplementation.
                    NamedPipeNotifier,
                    NamedPipeNotifyImplementation" />
      <register type="NotifyDefinitions.ISystemWatcher,
          NotifyDefinitions"
                mapTo="MiddlewareSystemWatcher.
                    LocalFIleWatcher, MiddlewareSystemWatcher"
                     />
    </container>
  </unity>
```

The first responsibility of the ATM is to react to context changes. Here we use a FileSystemWatcher which raises an event when the settings file is changed. The ATM handles event by calling a Notifier method on the NamedPipeServer and gets an instance of CurrentContext class which is passed to Notifier as an argument.

NamedPipe Notifier

The ATM uses a Notifier dependency to carry out communication to player wrappers. We use a NetNamedPipe connection with all wrappers so that only one Notifier instance is required. Another option would be to use a collection of Notifiers which implement different types of IPC and loop through each and call Notify() method.

There are three functional interfaces and implementations used in server-clients communication.

- INotifier interface which defines a contract for the service host and includes methods to start and stop server, and start callback notifications.
- IPlaybackController interface which defines contract for service endpoint, and includes methods for initializing and closing connection and method to return the settings.
- ICallback interface which defines a contract for the service client and includes methods that a service host can call when it needs to send new setting update.

NamedPipe server (INotifier implementation) uses a Windows Communication Foundation (WCF) ServiceHost to instantiate a communication server. It needs a baseURI address which in our case is
net.pipe : *//localhost/playbackserver/getsettings*.

Service endpoint (IPlaybackController implementation) includes binding setting **netNamedPipeBinding** and a relative ServiceURI to the server which is a string that needs to have the same value as the method for getting settings, i.e. *GetSettings*. This can be basically any string and its only important that both the server and the client use the same string value.

When the service host server starts, it loads a service endpoint which is ready to initialize connections to the clients. Every client implements ICallback interface which server can use to call a method when dispatching updates. Upon client initialization the service endpoint adds a callback reference to a dictionary collection with the callback object and the client name. Callbacks are essentially proxies to real client objects which .NET runtime can call and execute methods on. When service host gets a Notify message from the ATM it loops through a Callback collections and calls **DispachSettings** methods for every proxy in collection. We have included a client name for the proxy to identify if the same proxy/name combination has already been added. This is part of video-helper-lib logic to implement that every player has its own unique name.

WCF NamedPipe class can have many clients attached as explained in Subsection 3.3.2. We tested using it with three client and achieved equal performance as with a single client. The only measurable delay was looping trough the collection of connected clients with an average of 203 nanoseconds to move to the next item.

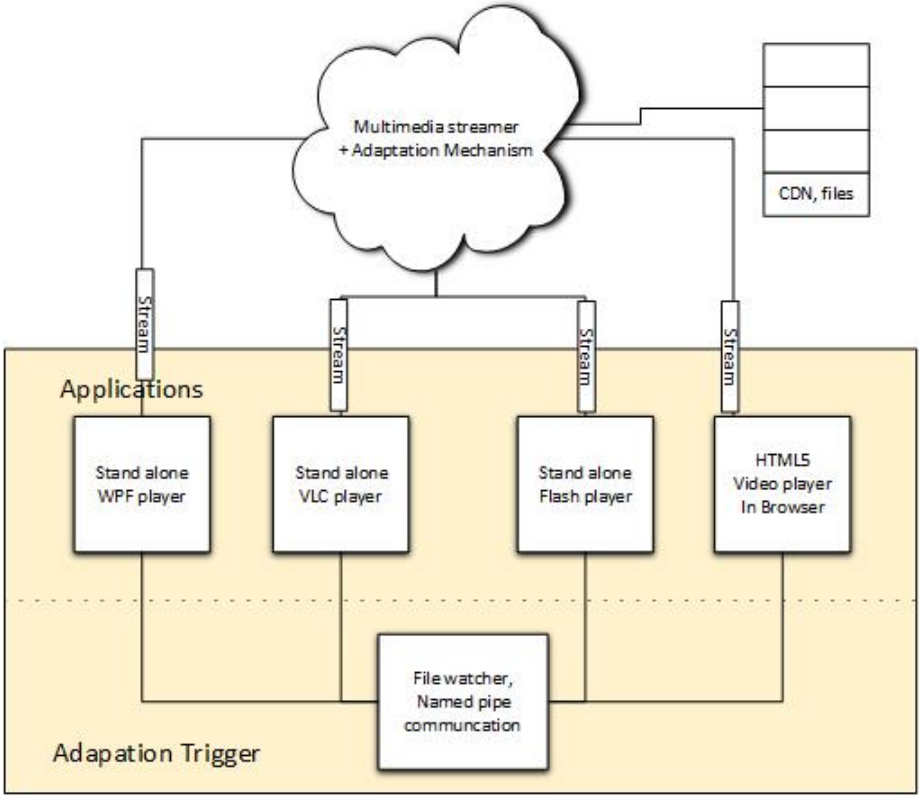
4.1.3 Video helper library

Each of presented player wrappers is using video-helper-lib to control a NamedPipe client which handles the communication with the NamedPipe server instantiated by the ATM. The video-helper-lib has implementations for two important interfaces:

- INamedPipeClient interface which serves as NamedPipe client and includes methods for starting and stopping the client, as well as calling GetSettings on the NamedPipe server.
- ICallback interface which has method and a delegate property used by the NamedPipe server to signal changes back to the client.

INamedPipeClient implementation opens a DuplexChannelFactory with the netNamedPipeBinding that uses the same URI endpoint address as the server. It uses ICallback implementation object that implements a callback contract. ICallback implementation has a method that invokes the delegate property with CurrentContext as argument. This Delegate object is implemented by each player in a special way. Flash player calls JavaScript API to change quality or mute, while WPF and VLC players call their own methods respectively.

Figure 4.6: Actual design.



To put calls in the perspective, GetSettings is the method call that starts the sequence presented in the sequence Figure 3.2, while DispatchSettings which calls delegate is the sequence presented in Figure 3.3.

4.2 Challenges discovered

The challenges discovered in implementation are regarding both players and multimedia content providers.

4.2.1 Players

Our initial design shown in Figure 3.1 is achievable with the use of the existing players. In all cases we have tested, the players had the direct link to the media source. The logic for media link couldn't be abstracted away and moved to a ATM to free the player wrappers of the responsibility to react to context changes. The design that we managed to achieve is shown in Figure 4.6, and actual sequence diagrams in figures 4.7 and 4.8.

In order for the ATM to implement the adaptation logic, we would need a player that uses components which can be separated from main logic and replaced as required. We realize that this is a complicated task to achieve as a player is made of many parts that need to work in accordance. Opening file, or a stream is hidden beneath layers of code,

Figure 4.7: Actual design - video player initiates stream.

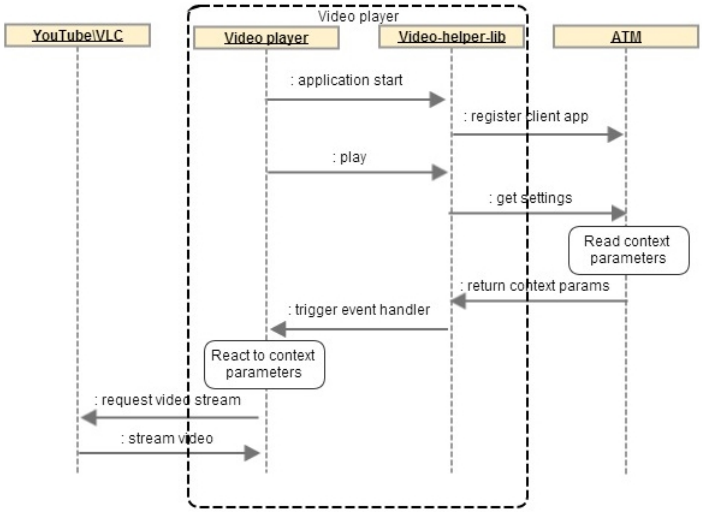
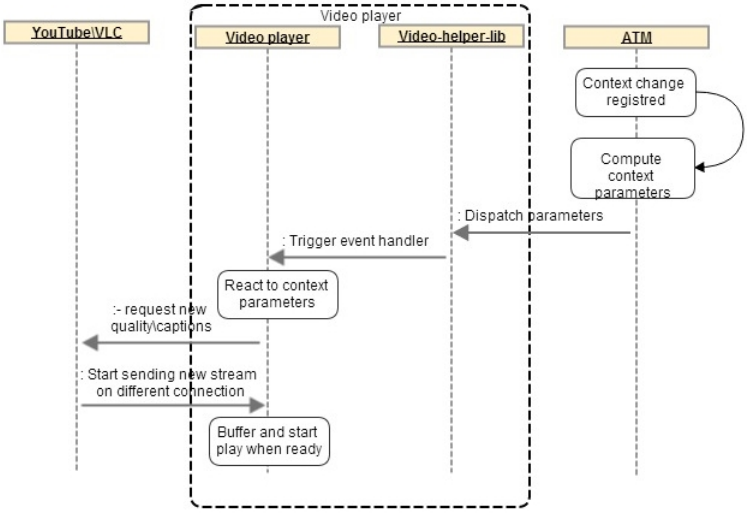


Figure 4.8: Actual design - context change initiates adaptation.



and connected to decoding, demuxing, memory management, buffering, preferences, skinning and many others.

4.2.2 Multimedia content providers

Both YouTube and other multimedia content providers offer very limited adaptation options. In fact YouTube does not offer on-the-fly adaptation at all. What YouTube does is a-priori transcoding of the video into several video formats with each having several video quality options. The formats are of FLV, MP4 and 3GP type aimed for different player versions in browser, or standalone players found on portable devices, TV sets and other devices. Upon requesting a video, YouTube sends the range of links in response, for different formats and bitrates. When the player needs to update the video quality, it uses a buffering mechanism that starts loading video with different bitrate in the background and switch to new video stream when enough data is ready in the buffer.

This might be "good enough" and the only feasible option for YouTube to deliver streaming to the large Internet audience they currently have.

A possible solution for dynamic adaptation is to create a video on demand server that has adaptation mechanism for fidelity, modality and content adaptation, which is also a demanding task.

4.3 Summary

We tried different approaches to overcome limitations with the players having direct link to the multimedia content and the YouTube adaptation mechanism.

1. - **Possible solution** Use VLC as an adaptation proxy and YouTube as a multimedia content provider. Then communicate the VLC stream url to the players and give command to render a stream.
   - **Result** Although VLC can adapt the video stream in many different ways, it cannot update the stream while it runs. Given the command to adapt the video stream again it starts to play from the beginning.

2. - **Possible solution** Manipulation of data at network level. Injection of packets in network to control adaptation. The YouTube issues a request for different bitrate so we tried to do the same.
   - **Result** YouTube player itself opens a new connection to request a new stream with different video quality. Injection of the command packet does not help in this case, since the injected packet is not affecting the player.

We also realize that since YouTube is not adapting the multimedia content at runtime we could not measure how fast it is adapting the stream. YouTube and other leading multimedia content providers are using buffering mechanisms to seamlessly deliver multimedia content and adapt

the video quality to match device and network conditions.

In the next chapter we look into how the server and the player can be simulated to make ATM include the logic for handling communication between the two.

Chapter 5

Simulation design and development

We saw in Section 4.2 that working with commercial players and multimedia servers introduce limitations for the ATM. The way they are designed affects the ATM not being able to handle adaptation parameters and work between the two.

In this chapter, we present a simulated server which simulates adaptation and a simulated player which simulate stream consumption where the video stream is requested and send by the ATM.

5.1 Design

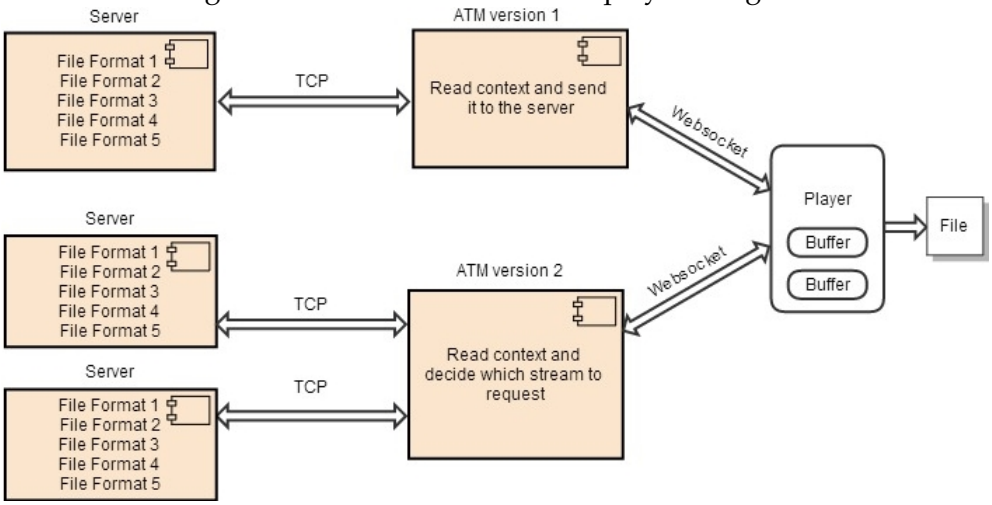
The design of the system is shown in Figure 5.1. There are two versions of ATM that we present. The way we want the server to work is to use existing open connection and send context parameters to the server which adapts the stream and continues streaming from the given position. The way YouTube works is that it provides links to different versions of the file and applications have to open a new connection to get the stream. This is shown respectivelly as ATM Version 1 and ATM Version 2 in Figure 5.1. The Figure shows Websocket for previously used named pipe which is explained in Subsection 5.2.3.

Considering that we want ATM to work with different versions of multimedia servers we investigate how ATM handles both cases.

5.1.1 Server design

To simulate both ATM versions we design a server that has a method for transfer of a video stream provided with the file name, context and position parameters for ATM version 1. The context parameter is used to return different file version, while position is used as a start position within the file. By context we denote the video quality of the stream.

Figure 5.1: Simulated server and player design



5.1.2 Player design

The simulated player in both cases is a stream reader that keeps a buffer for storing the stream and a current read position. When the player requests the stream it stores the stream data in a buffer. When the data arrives, the player starts reading from the buffer and dumping the data to a file. The stream is consumed by the reader which keeps the stream position parameter. This parameter is used when requesting for the stream adaptation. The player has another buffer used for storing the adapted stream.

5.1.3 File streamed

The file streamed is a video clip encoded in MP4 format. In Figure 5.1 we show five different file versions in different quality formats to illustrate a real scenario. Due to the fact that we are only simulating the server multimedia content adaptation here we can actually use the same file for returning in each version. Returning a different format and encoding of the file matters if we want to measure the quality of the adaptation aspect. This observation makes context parameter unnecessary but we keep it to illustrate a real scenario.

Another reason for using the same file is that stream "position" is relative to stream size as opposed to "time watched" which the server in fact requires. The stream object in the .NET does not support "length" property while being streamed, and to get relative position we need to know which file is being streamed, which adds a new parameter to the interface. This can be achieved by passing the file name, opening given file and measuring relative "time watched" by looking at the length and the current position. The measurement produces a percent number which gives a relative position in another stream. However this does not really

add any valuable information for the testing of the ATM. We measure response time and file size and encoding are not relevant in the simulated streaming. For these reasons the server response with same file in the calls with different context parameters.

5.2 Implementation

We implement both server and player as console applications. The server exposes an interface contract for communication which is shown in Listing 5.1

Listing 5.1: Simulated server interface

```
[ServiceContract(Namespace = "http://ATM.Stream")]
public interface IVideoStreaming
{
       [OperationContract]
       System.IO.Stream GetStream(string videoName,
           CurrentContext context, int streamPosition);
}
```

The parameter context have a type of CurrentContext which represents current device context. The values are not used in the simulation but for YouTube call this parameter would have Flash player variables (flashvar) name:value pairs. Position is integer which denotes a length in bytes from zero to current position within the stream.

Return type of the method is a Stream which is an abstract class in .NET [49]. It can be used for any type of stream and is used to encapsulate a file stream and a memory stream objects that we use in the implementation.

5.2.1 Server implementation

Server implements the IVideoStreaming interface by implementing the GetStream method which opens a file, assign it to a FileStream object, and set the current position from the provided parameter. If position is not set then number 0 is used to denote the start of the stream. The method returns the FileStream as a Stream super class.

The server endpoint is defined as in Listing 5.2. It uses the IP address of the server machine and the ports on the local area machine. The endpoint is a channel which multiple TCP sockets can use. Every player gets a dedicated TCP socket connection which is kept alive while streaming to the end.

Listing 5.2: Simulated server endpoint

```
<services>
    <service name="ATM.Stream.VideoStreamingService"
        behaviorConfiguration="NetTcpBehavior">
      <endpoint address="net.tcp://192.168.1.11:9090/
          TestService" binding="netTcpBinding"
          bindingConfiguration="NetTcpBinding_ITest"
          contract="Middleware.Stream.IVideoStreaming"/>
    </service>
  </services>
```

We use several servers to simulate YouTube behaviour of having different links for different files versions.

5.2.2 Player implementation

The player simulate consuming the stream by reading it from the buffer and dumping data to a file. The read method reads array of bytes from the buffer, where we can specify how much data is read in every read action. Standard value of byte array is set to 4KB at the time and decreasing this value simulates slower buffer reading. When buffer starts filling the Copy stream method is issued to copy the Stream to a FileStream which is saved to a disk. We keep position of the file by increasing value of it each time a byte array with specified buffer length is read.

The player wrapper uses the implementation of the wrapper described in Subsection 4.1.3. The player has a method to return a simulated stream position which is used in sending position to the server. All players use different names, together with a different file name requested. This is used in ATM with allocation of a dedicated websocket connection. The player name and the video file name are part of player configuration file. The reason for using websockets is explained in the next paragraph.

5.2.3 Inter-process Communication (IPC)

Windows Communication Foundation (WCF) can send messages in two modes: buffered and streamed mode. Microsoft explains on the web page [24] that: *"In the default buffered-transfer mode, a message must be completely delivered before a receiver can read it. In streaming transfer mode, the receiver can begin to process the message before it is completely delivered. The streaming mode is useful when the information that is passed is lengthy and can be processed serially. Streaming mode is also useful when the message is too large to be entirely buffered."*

The buffered transfer mode [53] is the default transfer mode and is supported by all binding types that exists in .NET framework. Streamed mode transfer is unreliable in the sense that the checksums are not applied at a whole message as one unity, rather on chunks of data. It is supported by

a subset of all bindings, such as BasicHTTBinding, NetNamePipeBinding, NetTcpBinding, NetHttpBinding, which represent HTTP transfer, named pipes, TCP sockets, WebSocket respectively. The only binding that supports Duplex mode and can have transfer mode streamed is NetHttpBinding which uses Websockets for bi directional streaming. The NetTCP binding is using request-reply streaming mode.
For these reasons we are using a NetTCP binding for communication between the server and the ATM and NetHTTP binding between the ATM and the player, as shown in Figure 5.1.

5.3 Changes in the ATM logic

Now that we have a player with a media source which can be controlled and a server which streams data directly to the ATM we can move context reaction logic to the ATM. This is the part of initial design shown in Figure 3.1 on the page 15. We keep the previous components for file listening, reading context parameters and communication as explained in Subsection 4.1.2.

For the reasons explained in previous subsection we add a new communication handler that implements INotifier interface but this time it uses NetHttpBinding as endpoint with URI address:
$net.tcp://localhost/playbackserver/getstream$.
This change implies use of Websockets in stead of previously used Named Pipes.

Since we have anticipated possible changes and build ATM to be easily extendible, the changes needed for adding Websockets are done in one place. We only need to add new INotifier implementation to the collection of Notifiers in the ATM which trigger Notify method when the context changes. The DI container loads this instance in stead of the previous Named pipe. The rest of communication interface is kept almost the same except for two additional changes. The video-helper-lib library which simulated player uses has to be extended to use NetHttp binding as well, and IPlaybackController interface (see Subsection 4.1.2) has a new method called $System.IO.StreamGetStream(int\ position)$ for getting a stream.

5.3.1 New challenges

There are several new challenges which we have not experienced when ATM didn't included context reacting logic:

1. Serving multiple clients - logic for processing streaming in parallel.

2. Getting stream position from the player while streaming - while stream is being consumed it occupies a channel and calling a method through a websocket needs to wait until the stream has been consumed.

3. Stream object challenges - Stream behaves as one big object.

4. Different server API's - the ATM should be able to handle multiple server version with multiple clients simultaneously.

1. Serving multiple clients

Dealing with this challenge introduces need for running calls in parallel. .Net runtime has three options for low level multi threading mechanism. They can be either **Thread**, **ThreadPool** or **Task**.

Thread is an actual OS-level thread which offers the highest level of control but is the most costly in terms of OS resources. It has its own memory stack and kernel resources. Creating threads manually is not always advisable because of the risk of too many threads created. In case with many threads, the processor has to perform many context switches between the threads, which produces an overhead when saving and reading the register state.

ThreadPool is essentially a wrapper a around pool of threads. It gives no control to which thread can run first but avoids overhead of creating too many threads. The ThreadPool does not give feedback on when the task has been done and is best used for a thread pool with short running threads.

The third option is a **Task** which lies at a higher abstraction level than the first two. Like the ThreadPool the task does not create a new thread automatically but does offer control when to start and gives result when it finishes. The latest .NET framework is optimized to use Task and provides a parallel library to easily control the tasks.

We use Task Factory object to chain three tasks which executes asynchronously when the target Task completes. Task chain can be initiated with a command:

```
Task.Factory.StartNew(getPositionFunc)
 .ContinueWith(antecendent => getStreamFunc(antecendent.Result))
 .ContinueWith(antecendent => pushStreamAction(antecendent.Result));
```

which starts three asynchronous task and adds them to the *TaskScheduler*. Running tasks asynchronously reduces contention between the tasks. *Task* uses native .NET delegates to execute methods calls. There are two types of native delegates use, an *Action* which just perform an action without returning a result, and a *Func* which is a delegate that returns a result. We use two *Func* delegates to get position and get stream, and one *Action* delegate to push the stream to a player.

- getPositionFunc — returns int result for the stream position
- getStreamFunc — returns a Stream object from the server sending position provided in previous *Task*
- pushStreamAction — pushes the Stream object to the player when previous *Task* returns

2. Getting stream position while player is consuming stream

When using *Transfermode*=streamed and consuming the stream while it is transferred, the websocket channels are occupied and block all consecutive calls that might be waiting from the ATM. We want to send a call to the player which stops a stream reader and returns a current stream position. Stopping the stream reader from within the player is easy where we just have to change one bool variable, but while the stream is being transferred we cannot use the same websocket connection. To get around this limitation we use a second websocket connection from ATM to the player. The second websocket is used to send a message to stop the player consuming a stream and return a current stream position after the reader stops. After stream the ATM receives the current stream position the websocket is used to push the adapted stream since the first websocket is being blocked by the previous Stream object — explained in more detail in the next subsection.

All websocket connections are identified by a player name value so we use convention of naming *player_x* and *stopplayer_x*, where *x* is a different number for every player, to send a stream and to stop reading. The websocket connections are paired in a container object, thus representing one single player, and used sequentially in the ATM.

3. Stream object

The Stream class [49] is an abstract class which can encapsulate any derived stream. We are using Filestream [18] when reading the file and sending it as a Stream over the TCP connection. The FileStream class can use resources like sockets and file handlers to access the file on the disk.

The FileStream object does not support a pause/resume option while transferring. Once the streaming has started, it has to be consumed to the end or aborted. Aborting the stream is not a good option because it closes the socket connection and sets the state of the channel to aborted, which blocks consecutive calls to the server until the server is restarted. The player can however stop to read the incoming stream when the context change and receive a new stream from the ATM, but the server continues sending a stream to the ATM anyway.

This observation was found at the end of the time for this thesis and we did not had the time to develop a solution. The same observation applies to other native IO Streams like BufferedStream and MemoryStream. One possible solution is to use a custom stream which formats a FileStream into chunks that can be separated by the server and put together by an ATM before sending to the player. Another possible solution is to use a Chunking Channel [11] instead of IClientChannel which we are using. We recommend looking into both options as a future improvement of the ATM.

Listing 5.3: IServerApiMapper Interface

```
public interface IServerApiMapper
```

```
{
    void StartServiceClient();
    void StopServiceClient();
    Stream GetStream(string videoName, CurrentContext
        context, int position);
}
```

4. Different server API's

Every commercial multimedia server has its own API which usually is different from the others. As example, YouTube uses a set of JavaScript API calls to control the player while VLC uses command line controls which are completely different. In addition to API difference, the languages differ in itself and we need both code wrapper from one language to another and the API mapper. When we have a set of *wrapped* API calls that C# can execute, we then need to map ATM calls to different multimedia servers API's. With the simulated server, we don't actually need a code wrapper since all code is written in C#. We can simulate that servers have a different API which requires a new interface called *IServerApiMapper*.

The *IServerApiMapper* implementation needs to know how to connect to the multimedia server and how to map the calls from ATM to the API of the server. To connect to the simulated server, we create a method stub for the Interface shown in Listing 5.1, which produces a class named *VideoServiceClient* with service endpoints and methods for reaching to the server. Then we create an implementation of *IServerApiMapper* which instantiates *VideoServiceClient* and is ready for receiving calls from the ATM. In the case of Simulated server we create a *SimulatedServerApiMapper* object which has a tight coupling to a specific method calls to *VideoServiceClient* class.

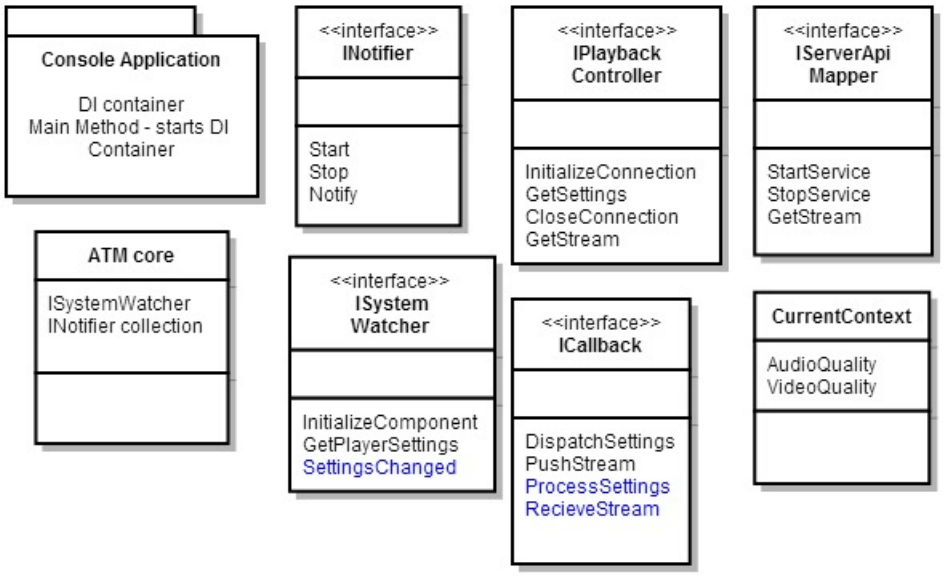
Every time a new player initiates a handshake, one implementation of *IServerApiMapper* is assigned to a Callback object which is added to the Callback collection. In our case it is above mentioned *SimulatedServerApiMapper*.

The interface in Listing 5.3 is simple and has only one method for getting a stream. That is because the simulated server and the player both contain one method in the testing of the ATM. This method can be extended to use other parameters as well, such as URL of the video coming from the simulated player. We imagine that when the real server API is available then it will grow and include that as well.

5.3.2 ATM component revised

To recap the revised main ATM components we show the class diagram of the main application and central interfaces that are implemented and loaded in the ATM in Figure 5.2 on page 51.

Figure 5.2: ATM main parts



1. **Console Application** is the ATM loader which loads ATM core library and has the main method which initiates a Unity DI container for loading dependencies.
2. **ATM core** starts a system watcher to watch for context changes and collection of notifiers, which can be either NamedPipe notifier of NetHttp notifier.
3. **ISystemWatcher** initializes the file change listener, and has a method for getting settings (or context), and an event that triggers when context is changed, called SettingChanged. ATM reacts to this event by calling Notify event for every notifier in collection.
4. **INotifier** has methods for starting and stopping host service and a notify which triggers message sending trough communication links.
5. **IPlaybackController** represents an ATM interface as seen from the client side. This interface is used by INotifier to start host service. It includes handshake method called InitalizeConnection, returning Settings to caller, Closing connection and GetStream when Player pulls the stream itself.
6. **ICallback** represent an interface for service client which the server sees and used for returning calls from a INotifier. DispatchSettings sends a context parameters, while PushStream pushes a stream from INotifier. The other two events, ProcessSettings and ReceiveStream trigger when each of the methods are called.
7. **IServerApiMapper** is used for mapping a logic from INotifier to a multimedia server API, as explained in this subsection.

8. **CurrentContext** is an object used for wrapping context parameters.

Chapter 6

Evaluation

In this chapter we describe the test bed, present the test results and lastly discuss the accumulated result sets.

6.1 Test

The simulation is run with two PC machines in the same LAN with the 802.11n wireless network. Both machines run Windows 8.1 OS. We call one of machines server machine and the other client machine. The server is a desktop PC which hosts the simulated streaming server, and the client machine is a laptop PC which hosts the ATM and one or more simulated players. The wireless access point can have a maximum bandwidth of 300 Mbps which is shared among the connected devices.

We use several files for streaming of size 64,3 MB. The files are video clips with length of 1:46 seconds encoded with H264 coded with bitrate 224 kb/s. All files are saved in the same folder on the server machine. The disk read speed is 40 Mbps. Files are copies of the same video clip where the only difference is the file name.

6.1.1 Test scenarios

There are several test scenarios which we run to benchmark ATM speed and performance.

1. One server with one player without context change — used for comparison
2. One server with one player with context change
3. One server with five players without context change
4. One server with five players with context change
5. One server with five players with context change where server returns a memory stream, in stead of a file stream
6. Two servers on different machines with one player with context change.

7. Two servers on different machines with five players with context change.

In the last two test scenarios we host one server on the server machine and in addition one server on the client machine. In both scenarios the ATM switches to use server on the client machine after the context change. The tests measure the TCP connection creation timespan with two servers and the response time with using different machines.

Each test is run 30 times to calculate an average timespan value. Adding more test rounds was unnecessary and shown the same timespan average. The time measured for a particular player starts from the context change event and ends when the player receives the first bytes from the new stream. The user can either start the streaming by pressing *Enter* in the console window of the player or the ATM can push the stream to the player from stream position 0. In the test with five players we trigger the initial streaming for all players with the context change event. Then we trigger context change again to start measuring the timespan.

Context change is triggered manually by changing a file from a text editor. We notice that different text editors change the file several times in a single file save. To prevent this from happening we pause file changed event while invoking SettingsChanged and resume it after. We still have noticed that some editors like Notepad++ [1] are changing file more then twice while saving. We use Microsoft Notepad in the tests which does not have this behaviour.

6.1.2 Benchmarking

There are three types of benchmarking that we use in testing the ATM:

1. testing adaptation trigger speed with the scenarios described in Subsection 6.1.1

2. testing performance metrics of the ATM, specified in the Requirements Section 3.2, and

3. testing code quality — test coding practices and complexity which ultimately result in more maintainable code.

Testing ATM speed

ATM reacting speed denotes the timespan from the time of a context change to the time a player receives an adapted stream. We noticed that each time we logged the time to text files a few milliseconds delay was produced, so logging is done directly to a console display and afterwards copied from the display and put together in a file.

[1] http://notepad-plus-plus.org/

Testing performance metrics

For performance metrics we use the profiling tools [44] available in the Visual Studio 2013 edition. The tools are called Performance and Diagnostic Hub, and are used to profile CPU usage and memory usage.

Testing code quality

A tool used for code quality analysis is NDepend [42] trial version, and the Visual Studio built-in code metrics tool.

6.2 ATM test results

In this section we present results for the tests described in Subsection 6.1.1.

6.2.1 Reaction speed results

The Console windows logs the timespan between the following events:

1. **ATM - Context change triggered** - time when the file listener registers a file change and triggers an event.
2. **ATM - Notify sent to client player** - time when the notifier object reacts to this event, which includes reading the file.
3. **ATM - Stream position returned** - time when the position is returned from the player.
4. **ATM - Send request to server from ATM** - time when the ATM sends request for adapted stream to the server.
5. **ATM - Stream received form server** - time when the adapted stream is returned to ATM from the server.
6. **Player - Adapted stream received** - time when the first byte of the adapted stream has reached the player.

We measure the time interval between the events described above to calculate the timespan values shown in the test result tables. The tables columns show the following timespan values in milliseconds:

Description — the description of the time interval

Mean value — average timespan value.

Median value — middle value in the list, i.e. the value that has half the values before and half after. In case when numbers are even, we take a sum of 2 in the middle of the list and divide by 2.

Mode value — the value that occurs most frequently in the list, i.e the number that is repeated more then any other number. If there is more then one mode then there is no mode for that numbers.

Range — minimum and maximum value.

Table 6.1: Results for one player **without** context change using the remote server

Description	Mean	Median	Mode	Range
Context changed event - Position sent to first client	0,93	0	0	0 - 22
Get a stream from the server to ATM	40,76	30	30	20 - 133
Send a stream further from ATM to a player	5,6	5	5	4 - 15
Total	47,3	35,5	36	25 - 167

Table 6.2: Results for one player **with** context change using the remote server

Description	Mean	Median	Mode	Range
Context changed event - Position sent to first client	0,76	0	0	0 - 19
Round-trip ATM - player - ATM to get stream position	6,6	5	5	1 - 41
Get a stream from the server to ATM	40,6	33,5	33	30 - 192
Send a stream further from ATM to a player	2,7	3	3	1 - 6
Total	50,8	43	N/A	34 - 258

Testing one player without context change using the remote server

The test results with one player are in shown in Table 6.1. In this case, the ATM initiates a stream transfer by sending a request to the server on the remote machine and push the stream to the player.

Testing one player with context change using the remote server

Test results with using one player after a context change happens with stream from the remote server are shown in Table 6.2. Here, we use the same server to transfer the second stream and use the second websocket connection to push the stream after the context change.

Testing five players without context change using the remote server

Table 6.3 shows results for testing ATM transfer of five streams from remote server to the five players.

Testing five players with context change using the remote server

Table 6.4 shows results with five players after the context change using the remote server.

Table 6.3: Results for five players **without** context change using the remote server

Description	Mean	Median	Mode	Range
Context changed event - Position sent to first client	0,96	0	0	0 - 23
Get a stream from the server to ATM	187,76	151,5	199	64 - 1490
Send a stream further from ATM to a player	7,9	7	7	2-24
Total	196.6	162	107	68 - 1519

Table 6.4: Results for five players **with** context change using the remote server

Description	Mean	Median	Mode	Range
Context changed event - Position sent to first client	0,83	0	0	0 - 19
Round-trip ATM - player - ATM to get stream position	9	7	7	2 - 50
Get a stream from the server to ATM	202	196,5	215	57 - 634
Send a stream further from ATM to a player	5,2	5	5	3 - 18
Total	217,1	208,5	131	66 - 710

Table 6.5: Results for five players **with** context change using the remote server returning memory stream

Description	Mean	Median	Mode	Range
Context changed event - Position sent to first client	0,8	0	0	0 - 18
Round-trip ATM - player - ATM to get stream position	9,3	8	8	1 - 42
Get a stream from the server to ATM	49,9	26	26	8 - 518
Send a stream further from ATM to a player	3,3	2	2	1 - 35
Total	63,48	37,5	34	16 - 578

Table 6.6: Results for one player **with** context change using the two servers

Description	Mean	Median	Mode	Range
Context changed event - Position sent to first client	0,5	0	0	0 - 15
Round-trip ATM - player - ATM to get stream position	4,9	5	5	2 - 5
Get a stream from the server to ATM	6,3	6	6	4 - 8
Send a stream further from ATM to a player	3,8	3	3	3 - 5
Total	15,2	15	15	10 - 33

Testing five players with context change using the remote server which returns memory stream

Table 6.5 shows results with five players after the context change using the remote server, only this time we return memory stream from memory cache, in stead of previously reading a file from the disk.

Testing one player with context change using the two servers

Table 6.6 shows results after the context change with one player. The initial data is requested from the remote server while second request is from the local server.

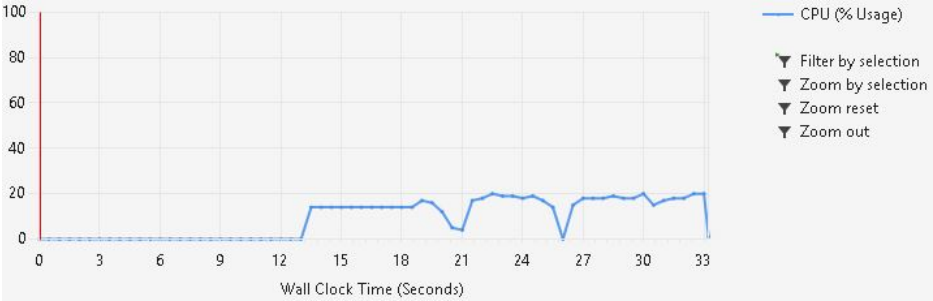
Testing five players with context change using two servers

Table 6.7 shows results after the context change with five players. Same as in previous test, the initial data is requested from the remote server while second request is from the local server.

Table 6.7: Results for five players **with** context change using two servers

Description	Mean	Median	Mode	Range
Context changed event - Position sent to first client	0,17	0	0	0 - 6
Round-trip ATM - player - ATM to get stream position	8,1	8	8	1 - 27
Get a stream from the server to ATM	18,9	18	18	2 - 49
Send a stream further from ATM to a player	7,85	8	8	1 - 17
Total	35,34	35,5	36	6 - 79

Figure 6.1: ATM CPU usage with file stream



6.2.2 Performance metrics results

The performance metrics results are obtained with the Visual Studio tool set Performance and Diagnostic Hub. The tools measure CPU and memory usage while adding players and reacting to context change with the rest of the logic included in the process.

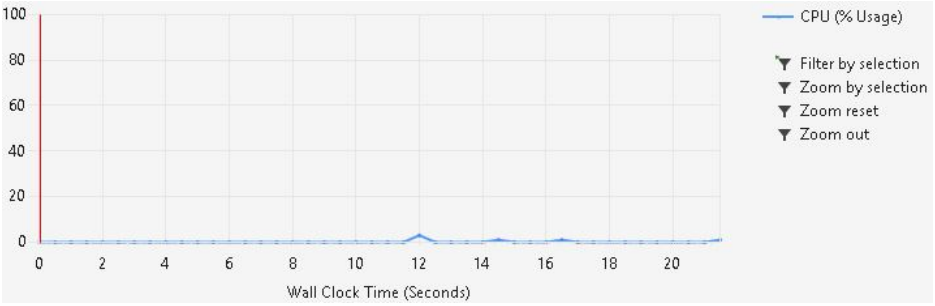
Figure 6.1 shows the ATM CPU percentage with the server using *Filestream* and the disk access and Figure 6.2 with the server using *Memorystream* without the disk access. The *Task* object, discussed in Subsection 5.3.1 includes a function call to get a stream from a server which adds the processing time of the server using disk access. We see that by adding five players to read simultaneously the CPU usage 20% while using memory stream generates maximum of 2% CPU usage. Using the stream from the remote server generates the same CPU usage percentage as with the memory stream.

The rest of screen shots generated are rather big and are included in the Appendix A.

Figure A.1 on page 80 shows percentage use of CPU for different processes within the ATM. It shows that the largest share of CPU resources is consumed by the Unity DI object instantiation process.

Figure A.2 on page 81 shows ATM memory usage. It is using a total of 12,1 MB at the most and primarily with allocating resources for the communication with players and server.

Figure 6.2: ATM CPU usage with memory stream



6.2.3 Code quality results

For the code quality matrix we use two tools, NDepend and Visual Studio built-in Code metrics. NDepend has helped us greatly to create more maintenable code by providing insight in tight coupling, resolving dependencies, potential complexity in methods and so on. We include a dependency matrix showing dependency distribution between the projects in the Figure A.3 on page 82. We have 12 projects divided in four major groups for Common, ATM related, Player wrappers, and Simulated apps, with dependencies shown in the Figure A.3. The top of the graph with yellowish background shows dependencies for our 12 projects while the bottom with blueish background shows .NET and COM dependencies. The project numbers, next to the name of the projects on the Y axis are used to represent the same projects on the top X axis. The box with the green color means that assembly in the column is used by the assembly in the row. The box with the blue color means that the assembly in the row is used by the assembly in the column. All boxes include value which represent the strength of the coupling. For example NotifyImplemntations have a value 3 in the column with the header 6. The number 6 in the column header represents a ServerApi assembly while the value 3 in the cell represent number of *coupling* or dependencies, i.e. NotifyImplementations is referencing ServerApi 3 times. They all share common interfaces with implementation abstracted away with the DI framework, so dependencies have to be there. Ndepend explains the layered code pattern: *"One pattern that is made obvious by a DSM is layered structure (i.e acyclic structure). When the matrix is triangular, with all blue cells in the lower-left triangle and all green cells in the upper-right triangle, then it shows that the structure is perfectly layered. In other words, the structure doesn't contain any dependency cycle"* [15].

NDepend has many features which help to uncover possible code problems, and create many useful graphs that can be overwhelming with details, so we used Visual Studio Code metrics to produce an easy understandable code metrics in Figure 6.3.

The most important column is the Maintainability Index which *"calculates an index value between 0 and 100 that represents the relative ease of maintaining the code. A high value means better maintainability"* [12]. The code below 20 is considered moderately maintainable, and below 10 not maintainable. We see that most of the project holds values over 70, and the ATM score 83.

Figure 6.3: ATM Code metrics

Hierarchy	Maintainability Ind...	Depth of Inheritance	Class Coupling
ATM\ATMCore (Debug)	83	1	7
ATM\ATMSimulatedPlayer (Debug)	85	1	39
ATM\ATMSystemWatcher (Debug)	72	1	23
ATM\ATMWindowsConsole (Debug)	78	1	7
Common\CommonArtifacts (Debug)	93	2	38
Common\NotifyImplementation (De	75	1	56
Common\PlayerCommLib (Debug)	90	1	19
Common\ServerApi (Debug)	78	1	8
MediaPlayers\WinFormFlashPlayer (D	66	7	70
MediaPlayers\WpfVLCplayer (Debug)	81	9	43
MediaPlayers\WpfYoutubePlayer (De	94	9	16
Simulated\FakeServer (Debug)	82	1	19

Depth of Inheritance should be kept low. The player wrappers have the highest level of dependence which originates from using many derivatives of media source classes.

Class coupling is high in some projects due to the fact that many resources are not abstracted, such as file readers, loggers, system delegates and such.

6.3 Discussion

6.3.1 Discussion of test results

Considering results shown in tables in the previous section, we see that some actions used similar timespans regardless of the number of players, while other had different timespans depending on which server version was used. In this subsection we examine each action in more detail.

Context change event shows the timespan from the file change event until the ATM has found the first callback-pair object in its collection. This action include the disk read of the file with context parameters. The average timespan value is 150 nanoseconds. It is only for the very first time when the ATM finds and reads the file in memory which takes maximum of 23 milliseconds.

Round-trip ATM - player - ATM shows the time the function call takes from the ATM to the player which first stops stream reader, gets the stream position and returns it back to ATM. The usual timespan range is from 5 to 8 ms depending on the number of players. We see in the range cell that time can be as long as 50 ms. This happens at the first time the connection from the ATM to the player is created which includes channel factory instantiation. Channel factory is discussed in the next paragraph.

Get a stream from the server to ATM shows the time for request-response round-trip from ATM to the server, where response is measured when the

first byte of adapted stream reaches the ATM thread. Pinging the remote server from local machine gave results in Listing 6.1.

Listing 6.1: Ping result client machine - remote server machine

```
Ping statistics for 192.168.1.132:
    Packets: Sent = 100, Received = 100, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 2ms, Maximum = 200ms, Average = 10ms
```

Here, we experienced most irregularities, but also some patterns. The timespan range can be from as low as 2 ms for a single player using a local server, and as long as 1490 ms using the remote server. We see that the more players we add to the same server the longer timespan is for the players with requests coming the last to the server. This is primarily with the case when the server reads the disk data. Comparing tables 6.4 and 6.5 shows the big difference where the first table has 202 ms and the second 49,9 ms for average time the server sends the stream. This is because the simultaneous disk reads have to share the disk resources such as disk read head. We see that the average time using memory stream is slightly higher then with the single player reading from the disk. This means that the server has other concurrency issues, such as sharing network hardware resources. This might be greatly improved if we had more server hosting machines.

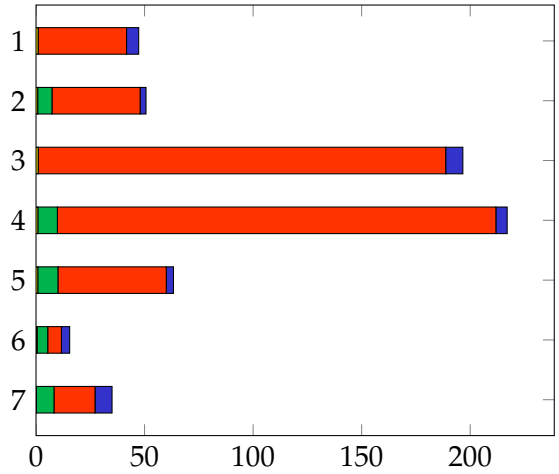
The timespan a single player uses is rather short and around 40,76 ms on average from the remote server. We see on several occasions that the server response time is longer then usual. This was noticed also for the first time we run test where the server needs to instantiate the connection.

The interesting case is for the ATM version 2, where a new socket was created for every time context changed. It took on average 30 ms for the first time to create a new socket, but after that, the timespan was 1.5 nanoseconds for creating a socket to the local server and 467 nanoseconds for creating a socket to the remote server. Examining this behaviour in detail shows that ChannelFactory class [9], which is a generic class for all service endpoints, is taking time to be instantiated and make ready. The Microsoft webpage explains why creating ChannelFactory has that overhead. *"Creating ChannelFactory<TChannel> instances incurs some overhead because it involves the following operations: Constructing the ContractDescription tree, Reflecting all of the required CLR types, Constructing the channel stack and Disposing of resources"* [8]. After instantiation .NET runtime is caching the factory and reusing it in subsequent calls. Creating a new socket was almost always 30 ms in the first call and this should be taken in consideration when migrating the player with .NET connection.

Send the stream further from ATM to the player shows the time the adapted stream will use NetHTTP channel to get from ATM to the player. Timespans are between 2 ms to 10 ms with with the average of 5 ms. This

timespan is the most steady and scale the same regardless of the number of the players added.

Total timespan ranges from 26 ms to 1519 ms in one single extreme case. Not counting the time for getting stream to server the average of all processes combined is 10,6 ms. If we add socket creation time for the first call, then the average goes up to 40,6 ms. It is still below the tolerable threshold of 100 ms, but with time added for getting the stream it might be long enough for a user to experience a problems with video delivery with maximum average of 217,1 ms. The below graph shows the average times used in the test where the Y axis is the test number as described in Subsection 6.1.1 and the X axis is the average time in ms.



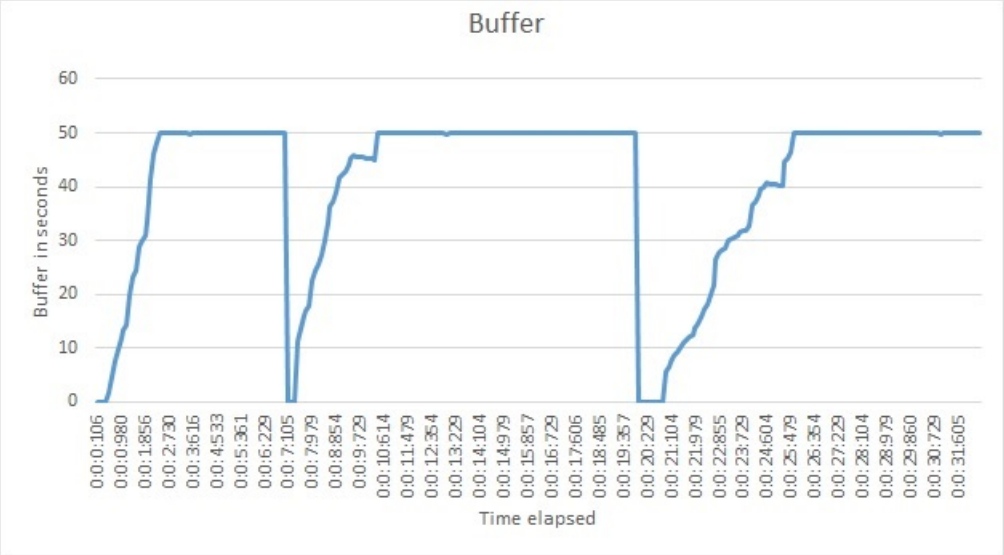
The colors represent:

- yellow - average time for the context changed event timespan,
- green - average time to get the stream position from the player,
- red - average time to get stream from the server, and
- blue - average time to send the stream to the player from the ATM.

We see that the red bar is the longest in most cases representing time to get a stream from the server. In the last two cases this timespan was negligible since the server was on the same machine as the player. However, longer waiting periods are possible if the server takes long time to send the stream. The remote server has average network delay of 10 ms.

This behaviour can be hidden from the user by introducing a second buffer to a player where the first one is used until there is enough data in the second one. This is also possible in our simulated player implementation to wait with stopping the stream reader until the second stream has been received through the second connection. This method is used by YouTube Flash player and is examined in the next subsection.

Figure 6.4: Dump from YoMo log.



6.3.2 Comparing reaction speed with YouTube

We mentioned briefly in Subsection 4.2.2 that the YouTube player is adapting the video quality by seamlessly transitioning from one stream to the other using a buffer mechanism. In this Subsection, we investigate how fast the YouTube player adapts the quality and compare it to our ATM reaction speed.

For measuring YouTube characteristics we use the YouTube monitoring tool called YoMo [48], designed and developed at the Informatics Department at University of Wuerzburg, Germany. YoMo is designed as a JavaScript plugin extension which logs the players runtime characteristic, and works in the Firefox browser. It logs data available by calling the YouTube JavaScript API. The data we are interested in is the buffer size represented in seconds of playback time. YoMo works with both Flash and HTML5 YouTube players. We used the Flash player in this evaluation. The part of the log file showing buffer size and time is shown in Figure 6.4.

The YouTube player has a change quality button which shows available video qualities, denoted by a bitrate and usually in the range from 140p to 1080p. When a user changes the quality in the Flash player control, the player continues to play the old stream and also begins to download the new quality stream in the background. That is done since it initially wants to fill up the buffer to a certain level for the new stream with the new quality.

Figure 6.4 is showing the values from the main stream buffer which includes the values for the new stream. It shows how much video playing time buffer has and the time it takes to fill it with the new quality stream. The two change quality events, where the line in the graph drops and goes back to 50 seconds are produced manually by clicking on the

change quality button. The YouTube reaction speed depends on many factors where the bandwidth is the most influential. Other factors like host characteristics and availability of the video take a small share of the reaction timespan.

The two change quality events in the graph illustrate the average time of filling the buffer which is in the range from 3 to 6 seconds. The graph is rather short to illustrate average value. Testing with more quality changes produces the same average value. On the test machines in the same network we experienced the average YouTube quality change timespan of around 5 seconds. While watching the video during quality change process, we have not experienced any flicker or interruptions.

The YouTube player on desktop machines does not react to bandwidth changes automatically while it does so on the portable devices. YoMo thus cannot measure how fast the YouTube player react to changes only how fast it fills the buffer after quality change.

We see that the response time from our server is much longer than what we experienced with YouTube . The YouTube player hides this behaviour with its buffering mechanism.

6.3.3 Discussion according to the requirements

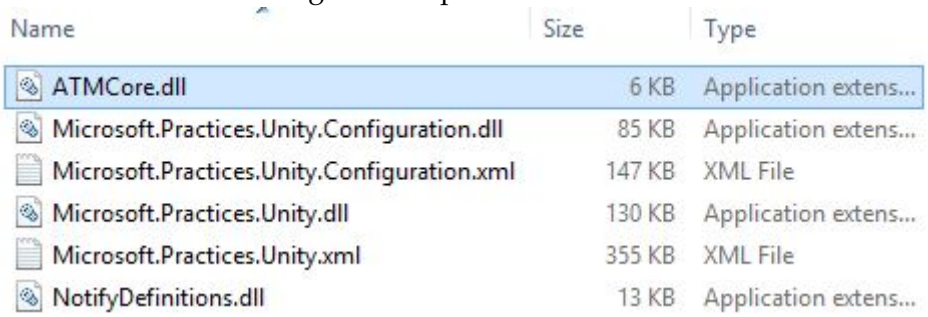
For the rest of this chapter, we evaluate ATM according to the requirements presented in Section 3.2.

Decoupled from applications: The ATM is running in its own process space and is not dependent on any other process except. Since it is a library by design, it has to be loaded by another process. We have used Windows console application to start it but it can run in other types as IIS web site or Windows form background application. The connection to other clients is handled by a communication component, i.e. named pipe or websocket server which is dynamically loaded.

Scalability: By using a named pipe and websocket connection types the ATM can be connected as many clients as the host machine can run. There is no upper limit dictated by the communication channels. Adding more players however increases the average ATM response time where the only sequential processing is looping through a collection of Callback pairs and starting a new Task that handles the rest of the process. The average timespan for this process has been measured to 203 nanoseconds. The average timespans for using communication objects between ATM and players are stable and scalable. It is only the server response time that does not scale well, but that is something that ATM cannot control.

Small footprint: Figure 6.5 shows space on the disk the ATM is using. ATMCore.dll file is using only 6 kilobytes, and the most of the space is used by Microsoft Unity Dependency Injection framework. We realize that it could use even less space if we use another DI framework or create our own

Figure 6.5: Space on the disk



dependency container. NotifyDefinitons library is a common library shared by all projects. Another important libraries ATMSystemWatcher.dll (11 Kb), NotifyImplementation.dll (18 Kb) and ServerAPI.dll (5 Kb) used for context listening and handling communication and are loaded dynamically by the DI framework.

Low resource consumption: We saw in Figure A.2 on page 81 that ATM is using total of 12,1 MB of memory.

Efficient: The results discussed in previous section showed that average timespan for all processes including generation of new socket is 40,6 ms. This value is efficient enough so the user does not experience drop in the QoE.

Open and extendible: The ATM is using Unity DI framework to load only libraries needed. The libraries for listening the context and handling communication are compiled in own dll files. They are loaded by a DI when Unity DI container process a configuration file. They can be changed easily by changing the setting in the file.

Support video streaming over internet: The ATM aids the players in providing context parameters which can be sent to players media source and use in adaptation. It cannot however control the stream of the tested legacy players.

Chapter 7

Conclusion

We cannot confirm neither of the initial assumptions described in Section 1.4. The legacy multimedia content providers, YouTube included, does not provide dynamic content adaptation and to the best of our knowledge we were not able to find any commercial vendor that does provide that. As for the players, the ones we used in test did not have a public API to control media. That had lead us to investigation of simulated streaming but still some questions remain open.

In this chapter, we give answer to the questions from the problem statement in Section 1.3. We discuss contributions and the future work and finish the thesis with the final words.

7.1 Conclusion

Question Can the ATM help legacy players by reacting to context and requesting adapted video stream for the players by keeping reaction time below tolerable delay?

Answer To Answer this question we tried to connect the ATM with traditional VoD players and multimedia streaming servers. The ATM is able to communicate with the players and control the playback but it is not able to include the logic for controlling the media source link (because the API not being open), thus making the player wrappers have this responsibility. In the simulated part, we are able to control the streaming. The average total reaction timespan is 10,6 milliseconds without initial socket creation, 40,6 ms including socket creation. Including different server response times, the total average is at maximum 217,1 ms. The amount of time a public CDN server takes to respond is expected to be much higher. The time the YouTube server takes to respond is around 5 seconds tested on our machines and we see that this is handled by using the second or more buffers to seamlessly switch the media stream.

So the answer to this question is still open. We think that it is possible to use ATM with a custom player project where the player components can be abstracted and controlled. ATM on Windows 8

reacts with good speed and fast enough to read the context and sent the notification calls to the clients.

Question Can the same ATM serve many different applications and achieve the same efficiency?

Answer The scalability issues we experienced with the ATM after adding more players were mostly in regards to server response time. If ATM is using different servers then the total time will be according to the server response times. Since all the calls are done in parallel the number of clients is depending on the host machine characteristics. So the answer to the question is yes, it can serve many different applications simultaneously.

Question Can we build ATM as easily extendible library while still achieve same efficiency?

Answer All the main ATM parts are abstracted with interfaces and loaded with the DI framework so changing is done in one place. We have achieved a good code review statistics as well. The answer to this question is yes.

7.2 Contributions

The work done in this thesis is a step towards understanding:

- the available multimedia adaptation mechanisms in popular VoD services,
- how to integrate adaptation mechanisms triggered by context changes to third party multimedia players and multimedia streaming servers

We have created a library that can be used to connect a multimedia server and a video player which other developers can build on further. We have investigated and developed code wrappers for one browser player and three stand alone video players, which can be used in the further research.

We also uncovered many limitations that can help in design and implementation of automatic context sensing and multimedia adaptation. Although the tools used were bound to Windows platform the code can be cross compiled without much effort and be used on other platforms.

7.3 Future work

Due to the lack of time for this thesis we leave out several parts which can be the topics for future work as a step towards ubiquitous consumption of multimedia.

Develop a player and a server ATM is evaluated with simple simulation of server and player, which gives us a valuable evaluation of it components speed and gives a good groundwork for building the application further. It will be very good to test it with real server that can adapt the stream before sending and use real player to show the stream. This way we could also have a qualitative study by measuring user feedback. We suggest looking into a player or a multimedia server that can be used for further evaluation of the ATM. A possible and promising platform for that is GStreamer developer platform.

Portability of the ATM. The current ATM is created in .NET which is by nature aimed for Windows platform despite projects like Mono which can port .NET code on other platforms.

However, there are many platforms that are widely used but still don't support .NET or Mono. One example is iOS which is very popular choice for smartphones and tablets. As a future work we suggest to use alternatives to .NET which could be an open HTML5 application designed to work with all native SDKs.

Context change sensing in ATM. We use a XML file to simulate context change. As a further work we suggest for ATM to use a native host API to sense device characteristics. With Windows it can be done by querying available device drivers while with Android it will be native call to Android SDK.

Use custom stream and custom formatter This suggestion is improvement of the current simulated server state which we had time to develop. The Stream object is blocking the callback proxy while the stream is sent to the end of file. This behaviour has an impact on ATM which has to use second connection to the player. Extending the Filestream to use custom formatting in file chunks will allow the server to stop the stream. Suggested solution is to use this with the Chunking channel type.

7.4 Final words

The work in this thesis has had several changes during its course. The initial intentions was to develop a software component in Java which can be cross compiled and tested on Android systems. This is why we included work on Spring MVC and Java websockets server in the HTML5 player wrapper. We saw that the progress was not going fast enough so we switched to using the familiar .NET framework. Using .NET has greatly increased our development but still there are many parts of the framework that can be even more investigated, particularly WCF.

Another change was when we found out that we were not able to use popular video players and VoD systems. The software component was initially intended to work in-between the two and the test would measure

how fast the ATM was reacting and how fast the multimedia content was adapting and what impact it has on the network traffic. Not being able to do so it has led us to simulate the server and the player and investigate more into inner workings of ATM. If we knew this limitations in advance we would try to invest more time into creating a custom server and a custom client application. On the other hand our findings turned out to be interesting for further investigation of streaming on the Windows platform and the state we leave the ATM would be easy to pick up and build on.

Bibliography

[1] *Adobe Flash Player*. Mar. 2014. URL: http://www.adobe.com/products/flashplayer.html.

[2] Velibor Adzic, Hari Kalva and Borko Furht. 'A survey of multimedia content adaptation for mobile devices'. In: *Multimedia Tools and Applications* 51.1 (2011), pp. 379–396.

[3] Pablo Ameigeiras et al. 'Analysis and modelling of YouTube traffic'. In: *Transactions on Emerging Telecommunications Technologies* 23.4 (2012), pp. 360–377.

[4] Davide Bellinzona and Patrick Vitali. 'Multimedia content adaptation through tag libraries'. In: *Database and Expert Systems Application, 2008. DEXA'08. 19th International Workshop on*. IEEE. 2008, pp. 716–720.

[5] Girma Berhe, Lionel Brunie and Jean-Marc Pierson. 'Content Adaptation in distributed multimedia system'. In: *Journal of Digital Information Management* 3.2 (2005), p. 95.

[6] Girma Berhe, Lionel Brunie and Jean-Marc Pierson. 'Modeling service-based multimedia content adaptation in pervasive computing'. In: *Proceedings of the 1st conference on Computing frontiers*. ACM. 2004, pp. 60–69.

[7] Michele Leroux Bustamante. *Learning WCF: A Hands-on Guide*. O'Reilly Media, Inc., 2007.

[8] *Channel Factory and Caching, Microsoft Developer Network*. May 2014. URL: http://msdn.microsoft.com/en-us/library/hh314046(v=vs.110).aspx.

[9] *ChannelFactory, Microsoft Developer Network*. Apr. 2014. URL: http://msdn.microsoft.com/en-us/library/ms576132(v=vs.110).aspx.

[10] Hui Chen, Bing Luo and Weisong Shi. 'Anole: a case for energy-aware mobile application design'. In: *Parallel Processing Workshops (ICPPW), 2012 41st International Conference on*. IEEE. 2012, pp. 232–238.

[11] *Chunking Channel, Microsoft Developer Network*. May 2014. URL: http://msdn.microsoft.com/en-us/library/aa717050.aspx.

[12] *Code Metrics Values, Microsoft Developer Network*. May 2014. URL: http://msdn.microsoft.com/en-us/library/bb385914.aspx.

[13] *Comet programming model*. Apr. 2014. URL: http://en.wikipedia.org/wiki/Comet_(programming).

[14] *Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies 1.0*. Feb. 2014. URL: http://www.w3.org/TR/CCPP-struct-vocab/.

[15] *Dependency Structure Matrix, NDepend*. May 2014. URL: http://www.ndepend.com/doc_matrix.aspx.

[16] *Desktop Operating System Market Share*. Mar. 2014. URL: http://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0.

[17] Anind K Dey. 'Understanding and using context'. In: *Personal and ubiquitous computing* 5.1 (2001), pp. 4–7.

[18] *FileStream Class, Microsoft Developer Network*. May 2014. URL: http://msdn.microsoft.com/en-us/library/system.io.filestream(v=vs.110).aspx.

[19] Alessandro Finamore et al. 'Youtube everywhere: Impact of device and infrastructure synergies on user experience'. In: *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*. ACM. 2011, pp. 345–360.

[20] Martin Fowler. *Inversion of control containers and the dependency injection pattern*. 2004.

[21] Xiaohui Gu and Klara Nahrstedt. 'Dynamic QoS-aware multimedia service configuration in ubiquitous computing environments'. In: *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*. IEEE. 2002, pp. 311–318.

[22] Majed Haddad et al. 'A survey on YouTube streaming service'. In: *Proceedings of the 5th International ICST Conference on Performance Evaluation Methodologies and Tools*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). 2011, pp. 300–305.

[23] Hans Vatne Hansen et al. 'Migration of fine-grained multimedia applications'. In: *Proceedings of the Posters and Demo Track*. ACM. 2012, p. 12.

[24] *How to: Enable Streaming, .NET*. Mar. 2014. URL: http://msdn.microsoft.com/en-us/library/ms789010(v=vs.110).aspx.

[25] *HTML5, A vocabulary and associated APIs for HTML and XHTML*. Feb. 2014. URL: http://www.w3.org/TR/html5/.

[26] *HTML5 elements video*. Apr. 2014. URL: http://www.w3.org/wiki/HTML/Elements/video.

[27] *HTML5 video*. Mar. 2014. URL: http://en.wikipedia.org/wiki/Open_video.

[28] *Interprocess Communicatins in Windows*. Mar. 2014. URL: http://msdn.microsoft.com/en-us/library/windows/desktop/aa365574(v=vs.85).aspx.

[29] Mohd Norasri Ismail, Rosziati Ibrahim and Mohd Farhan Md Fudzee. 'A survey on content adaptation systems towards energy consumption awareness'. In: *Advances in Multimedia* 2013 (2013), p. 3.

[30] Telecommunication Standardization Sector Of ITU. *ITU-T Recommendation G. 114-One-way transmission time*. 2003.

[31] Ramesh Jain and Pinaki Sinha. 'Content without context is meaningless'. In: *Proceedings of the international conference on Multimedia*. ACM. 2010, pp. 1259–1268.

[32] Svetlana Kim and Yong-Ik Yoon. 'Universal Multimedia Access Model for Video Delivery'. In: *Grid and Pervasive Computing Workshops, 2008. GPC Workshops' 08. The 3rd International Conference on*. IEEE. 2008, pp. 201–205.

[33] Tomas Kupka et al. 'Analysis of a Real-world HTTP Segment Streaming Case'. In: *Proceedings of the 11th European Conference on Interactive TV and Video*. ACM, 2013, pp. 75–84. ISBN: 978-1-4503-1951-5.

[34] Zhijun Lei and Nicolas D Georganas. 'Context-based media adaptation in pervasive computing'. In: *Electrical and Computer Engineering, 2001. Canadian Conference on*. Vol. 2. IEEE. 2001, pp. 913–918.

[35] Klaus Leopold, Dietmar Jannach and Hermann Hellwagner. 'A knowledge and component based multimedia adaptation framework'. In: *Multimedia Software Engineering, 2004. Proceedings. IEEE Sixth International Symposium on*. IEEE. 2004, pp. 10–17.

[36] *libVLC*. Nov. 2013. URL: https://wiki.videolan.org/LibVLC/.

[37] Juval Lowy. *Programming WCF services*. O'Reilly Media, Inc., 2007.

[38] *March 2014 Market Share*. Mar. 2014. URL: https://www.w3counter.com/globalstats.php.

[39] Rakesh Mohan, John R. Smith and Chung-Sheng Li. 'Adapting multimedia internet content for universal access'. In: *Multimedia, IEEE Transactions on* 1.1 (1999), pp. 104–114.

[40] A Moldovan and Cristina Hava Muntean. 'Subjective assessment of bitdetect—a mechanism for energy-aware multimedia content adaptation'. In: *Broadcasting, IEEE Transactions on* 58.3 (2012), pp. 480–492.

[41] *MyToolkit , Codeplex*. Feb. 2014. URL: https://mytoolkit.codeplex.com/.

[42] *NDepend, CODER IMPACT SAS*. May 2014. URL: http://www.ndepend.com/.

[43] *NetNamedPipeBinding Class on MSDN*. Mar. 2014. URL: http://msdn.microsoft.com/en-us/library/system.servicemodel.netnamedpipebinding.aspx.

[44] *Performance and Diagnostics Hub in Visual Studio 2013*. May 2014. URL: http://blogs.msdn.com/b/visualstudioalm/archive/2013/07/12/performance-and-diagnostics-hub-in-visual-studio-2013.aspx.

[45] Haakon Riiser et al. 'A comparison of quality scheduling in commercial adaptive http streaming solutions on a 3g network'. In: *Proceedings of the 4th Workshop on Mobile Video*. ACM. 2012, pp. 25–30.

[46] *Sandwine - Global Internet Phenomena Report*. Nov. 2013. URL: https://www.sandvine.com/downloads/general/global-internet-phenomena/2013/2h-2013-global-internet-phenomena-report.pdf.

[47] John R Smith, Rakesh Mohan and Chung-Sheng Li. 'Scalable multimedia delivery for pervasive computing'. In: *Proceedings of the seventh ACM international conference on Multimedia (Part 1)*. ACM. 1999, pp. 131–140.

[48] Barbara Staehle et al. 'YoMo: a Youtube application comfort monitoring tool'. In: *QoEMCS EuroITV* (2010).

[49] *Stream class, .NET*. Mar. 2014. URL: http://msdn.microsoft.com/en-us/library/system.io.stream.aspx.

[50] *TcpListener Class, .NET 4.5*. Apr. 2014. URL: http://msdn.microsoft.com/en-us/library/vstudio/system.net.sockets.tcplistener.

[51] Christian Timmerer et al. 'Guest Editorial Adaptive Media Streaming'. In: *Selected Areas in Communications, IEEE Journal on* 32.4 (2014), pp. 681–683.

[52] Ming-Wen Tong, Zong-Kai Yang and Qing-Tang Liu. 'A novel model of adaptation decision-taking engine in multimedia adaptation'. In: *Journal of Network and Computer Applications* 33.1 (2010), pp. 43–49.

[53] *TransferMode enumeration, Microsoft Developer Network*. Apr. 2014. URL: http://msdn.microsoft.com/en-us/library/system.servicemodel.transfermode.aspx.

[54] *Unity 3, Microsoft Developer Network*. Apr. 2014. URL: http://msdn.microsoft.com/en-us/library/dn170416.aspx.

[55] *Usage share of operating systems*. Mar. 2014. URL: http://en.wikipedia.org/wiki/Usage_share_of_operating_systems.

[56] *VideoLAN DotNET for WinForms, WPF and Silverlight*. Apr. 2014. URL: http://vlcdotnet.codeplex.com/.

[57] Bing Wang et al. 'Multimedia streaming via TCP: an analytic performance study'. In: *Proceedings of the 12th annual ACM international conference on Multimedia*. ACM. 2004, pp. 908–915.

[58] *WebSocket*. Apr. 2014. URL: http://www.websocket.org/index.html.

[59] *Windows 8 Store App development*. Apr. 2014. URL: http://msdn.microsoft.com/en-us/windows/apps/br229516.aspx.

[60] *Windows Communication Foundation, Microsoft Developer Network*. Feb. 2014. URL: http://msdn.microsoft.com/en-us/library/ms731082(v=vs.110).aspx.

[61] *WPF MediaElement Class*. Mar. 2014. URL: http://msdn.microsoft.com/en-us/library/system.windows.controls.mediaelement(v=vs.110).aspx.

[62] *Xamarin Studio*. Apr. 2014. URL: https://xamarin.com/.

[63] *YouTube from Wikipedia*. Feb. 2014. URL: http://en.wikipedia.org/wiki/YouTube.

[64] *YouTube Player API Reference for iframe Embeds*. Apr. 2014. URL: https://developers.google.com/youtube/iframe_api_reference.

Appendices

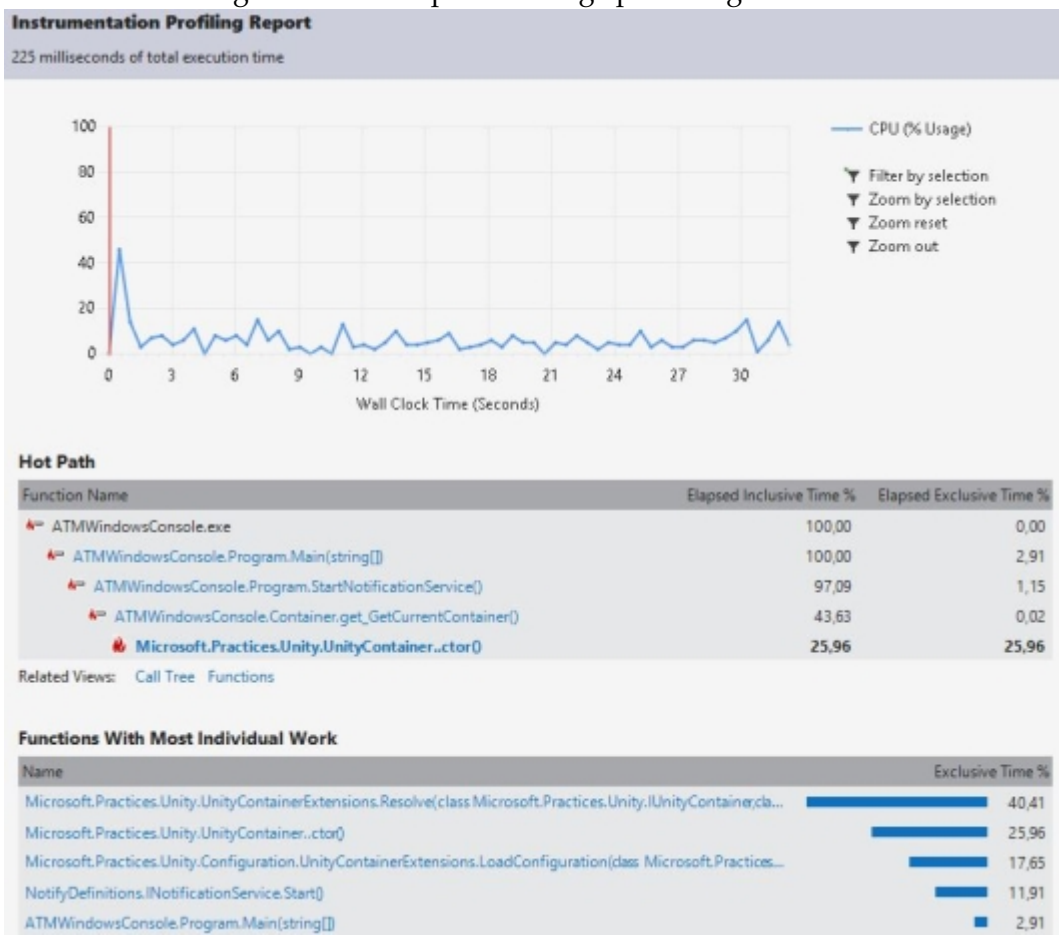
Appendix A

Screen dumps

Some figures were too big to include in the report, and are referenced and found in this appendix.

A.1 Performance Figures

Figure A.1: ATM process usage percentage



Instrumentation Profiling Report

225 milliseconds of total execution time

Hot Path

Function Name	Elapsed Inclusive Time %	Elapsed Exclusive Time %
ATMWindowsConsole.exe	100,00	0,00
ATMWindowsConsole.Program.Main(string[])	100,00	2,91
ATMWindowsConsole.Program.StartNotificationService()	97,09	1,15
ATMWindowsConsole.Container.get_GetCurrentContainer()	43,63	0,02
Microsoft.Practices.Unity.UnityContainer..ctor()	**25,96**	**25,96**

Related Views: Call Tree Functions

Functions With Most Individual Work

Name	Exclusive Time %
Microsoft.Practices.Unity.UnityContainerExtensions.Resolve(class Microsoft.Practices.Unity.IUnityContainer,cla...	40,41
Microsoft.Practices.Unity.UnityContainer..ctor()	25,96
Microsoft.Practices.Unity.Configuration.UnityContainerExtensions.LoadConfiguration(class Microsoft.Practices...	17,65
NotifyDefinitions.INotificationService.Start()	11,91
ATMWindowsConsole.Program.Main(string[])	2,91

Figure A.2: ATM memory allocation

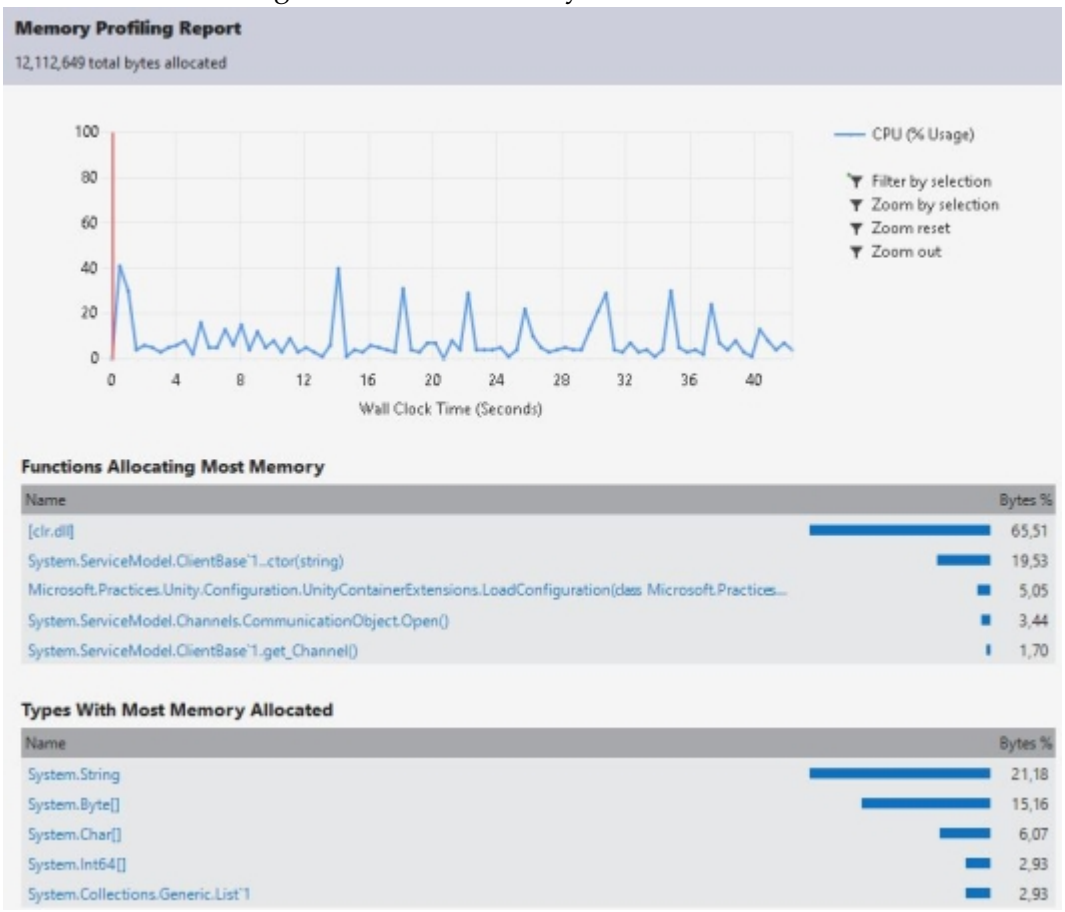


Figure A.3: ATM dependency graph